

# Parallel and Distributed Programming

Using

# C++

CAMERON  
HUGHES

TRACEY  
HUGHES

## Parallel and Distributed Programming Using C++

By [Cameron Hughes](#), [Tracey Hughes](#)

Start Reading ▶

Publisher: Addison Wesley

Pub Date: August 25, 2003

ISBN: 0-13-101376-9

Pages: 720

Parallel and Distributed Programming Using C++ provides an up-close look at how to build software that can take advantage of multiprocessor computers. Simple approaches for programming parallel virtual machines are presented, and the basics of cluster application development are explained. Through an easy-to-understand overview of multithreaded programming, this book also shows you how to write software components that work together over a network to solve problems and do work.

Parallel and Distributed Programming Using C++ provides an architectural approach to parallel programming for computer programmers, software developers, designers, researchers, and software architects. It will also be useful for computer science students.

## Copyright

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales  
(800) 382-3419  
[corpsales@pearsontechgroup.com](mailto:corpsales@pearsontechgroup.com)

For sales outside of the U.S., please contact:

International Sales  
(317) 581-3793  
[international@pearsontechgroup.com](mailto:international@pearsontechgroup.com)

Visit Addison-Wesley on the Web: [www.awprofessional.com](http://www.awprofessional.com)

A CIP catalog record for this book can be obtained from the Library of Congress.

Copyright © 2004 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.  
Rights and Contracts Department  
75 Arlington Street, Suite 300  
Boston, MA 02116  
Fax: (617) 848-7047

Text printed on recycled paper

1 2 3 4 5 6 7 8 9 10 First printing, June 2003

## **Dedication**

This book is dedicated to all the code warriors, white hat hackers, midnight engineers, and countless volunteers who have tirelessly given their skill, talent, experience, and time to make the open-source movement a reality and the Linux revolution possible. Without their tremendous contributions, the software needed to explore cluster programming, MPP programming, SMP programming, and distributed programming would simply not be as widely accessible and available to everyone in the world as it now is.

## **Preface**

We present an architectural approach to distributed and parallel programming using the C++ language. Particular attention is paid to how the C++ standard library, algorithms, and container classes behave in distributed and parallel environments. Methods for extending the C++ language through class libraries and function libraries to accomplish distributed and parallel programming tasks are explained. Emphasis is placed on how C++ works with the new POSIX and Single UNIX standards for multithreading. Combining C++ executables with other language executables to achieve multilingual solutions to distributed or parallel programming problems is also discussed. Several methods of organizing software that support parallel and distributed programming are introduced.

We demonstrate how to remove the fundamental obstacles to concurrency. The notion of emergent parallelization is explored. Our focus is not on optimization techniques, hardware specifics, performance comparisons, or on trying to apply parallel programming techniques to complex scientific or mathematical algorithms; rather, on how to structure computer programs and software systems to take advantage of opportunities for parallelization. Furthermore, we acquaint the reader with a multiparadigm approach to solving some of the problems that are inherent with distributed and parallel programming. Effective solutions to these problems often require a mix of several software design and engineering approaches. For instance, we deploy object-oriented programming techniques to tackle data race and synchronization problems. We use agent-oriented architectures to deal with multi-process and multithread management. Blackboards are used to minimize communication issues. In addition to object-oriented, agent-oriented, and AI-oriented programming, we use parameterized programming to implement generalized algorithms that are suitable where concurrency is required. Our experience with the development of software of all sizes and shapes has led us to believe that successful software design and implementation demands versatility. The suggestions, ideas, and solutions we present in this book reflect that experience.

## **The Challenges**

There are three basic challenges to writing parallel or distributed programs:

1. Identifying the natural parallelism that occurs within the context of a problem domain.
2. Dividing the software appropriately into two or more tasks that can be performed at the same time to accomplish the required parallelism.
3. Coordinating those tasks so that the software correctly and efficiently does what it is supposed to do.

These three challenges are accompanied by the following obstacles to concurrency:

Data race	Deadlock detection
Partial failure	Latency
Deadlock	Communication failures
Termination detection	Lack of global state
Multiple clock problem	Protocol mismatch
Localized errors	Lack of centralized resource allocation

This book explains what these obstacles are, why they occur, and how they can be managed.

Finally, several of the mechanisms we use for concurrency use TCP/IP as a protocol. Specifically the MPI (Message Passing Interface) library, PVM (Parallel Virtual Machine) library, and the MICO (CORBA) library. This allows our approaches to be used in an Internet/Intranet environment, which means that programs cooperating in parallel may be executing at different sites on the Internet or a corporate intranet and communicating through message passing. Many of the ideas serve as foundations for infrastructure of Web services. In addition to the MPI and PVM routines, the CORBA objects we use can communicate from different servers across the Internet. These components can be used to provide a variety of Internet/intranet services.

## **The Approach**

We advocate a component approach to the challenges and obstacles found in distributed and parallel programming. Our primary objective is to use framework classes as building blocks for concurrency. The framework classes are supported by object-oriented mutexes, semaphores, pipes, and sockets. The complexity of task synchronization and communication is significantly reduced through the use of interface classes. We deploy agent-driven threads and processes to facilitate thread and process management. Our primary approach to a global state and its related problems involve the use of blackboards. We combine agent-oriented and object-oriented architectures to accomplish multiparadigm solutions. Our multiparadigm approach is made possible using the support C++ has for object-oriented programming, parameterized programming, and structured programming.

## **Why C++?**

There are C++ compilers available for virtually every platform and operating environment. The ANSI (American National Standards Institute) and ISO (International Standard Organization) have defined standards for the C++ language and its library. There are robust open-source implementations as well as

commercial implementations of the language. The language has been widely adopted by researchers, designers, and professional developers around the world. The C++ language has been used to solve problems of all sizes and shapes from device drivers to large-scale industrial applications. The language supports a multiparadigm approach to software development and libraries that add parallel and distributed programming capabilities are readily available.

## **Libraries for Parallel and Distributed Programming**

The MPICH, an implementation of MPI, the PVM library, and the Pthreads (POSIX Threads) library, are used to implement parallel programming using C++. MICO, a C++ implementation of the CORBA standard, is used to achieve distributed programming. The C++ Standard Library, in combination with CORBA and the Pthreads library, provides the support for agent-oriented and blackboard programming concepts that are discussed in this book.

## **The New Single UNIX Specification Standard**

The new Single UNIX Specification Standard, Version 3, a joint effort between IEEE and the Open Group, was finalized and released in December 2001. The new Single UNIX Specification encompasses the POSIX standards and promotes portability for application programmers. It was designed to give software developers a single set of APIs to be supported by every UNIX system. It provides a reliable road map of standards for programmers who need to write multitasking and multithreading applications. In this book we rely on the Single UNIX Specification Standard for our discussions on process creations, process management, the Pthreads library, the new `posix_spawn()` routines, the POSIX semaphores, and FIFOs. [Appendix B](#) in this book contains excerpts from the standard that can be used as a reference to the material that we present.

## **Who is This Book For?**

This book is written for software designers, software developers, application programmers, researchers, educators, and students who need an introduction to parallel and distributed programming using the C++ language. A modest knowledge of the C++ language and standard C++ class libraries is required. This book is not intended as a tutorial on programming in C++ or object-oriented programming. It is assumed that the reader will have a basic understanding of object-oriented programming techniques such as encapsulation, inheritance, and polymorphism. This book introduces the basics of parallel and distributed programming in the context of C++.

## **Development Environments Supported**

The examples and programs presented in this book were developed and tested in the Linux and UNIX environments, specifically with Solaris 8, AIX, and Linux (SuSE, Red Hat). The PVM and MPI code was developed and tested on a 32-node Linux-based cluster. Many of the programs were tested on Sun Enterprise 450s. We used Sun's C++ Workshop, The Portland Group's C++ compiler, and GNU C++. Most examples will run in both the UNIX and Linux environments. In the cases where an example will not run in both environments, this fact is noted in the Program Profiles that are provided for all the complete program examples in the book.

## Ancillaries

### UML Diagrams

Many of the diagrams in this book use the UML (Unified Modeling Language) standard. In particular, activity diagrams, deployment diagrams, class diagrams, the state diagrams are used to describe important concurrency architectures and class relationships. Although a knowledge of the UML is not necessary, familiarity is helpful. [Appendix A](#) contains an explanation and description of the UML symbols and language that we use in this book.

### Program Profiles

Each complete program in the book is accompanied by a program profile. The profile will contain implementation specifics such as headers required, libraries required, compile instructions, and link instructions. The profile also includes a Notes section that will contain any special considerations that need to be taken when executing the program. Code that is not accompanied by a profile is meant for exposition purposes only.

### Sidebars

We made every attempt to stay away from notation that is too theoretical for a introductory book such as this one. However, in some cases the theoretical or mathematical notation was unavoidable. In those cases we use the notation but we provide a detailed explanation of the notation in a sidebar.

### Testing and Code Reliability

Although all examples and applications in this book were tested to ensure correctness, we make no warranties that the programs contained in this book are free of defects or error, are consistent with any particular standard or merchantability, or will meet your requirement for any particular application. They should not be relied upon for solving problems whose incorrect solution could result in injury to person or loss of property. The authors and publishers disclaim all liability for direct or consequential damages resulting from your use of the examples, programs, or applications present in this book.

### Acknowledgments

We could not have successfully pulled this project off without the help, suggestions, constructive criticisms, and resources of many of our friends and colleagues. In particular, we would like to thank Terry Lewis and Doug Johnson from OSC (Ohio Super-Computing) for providing us with complete access to a 32-node Linux-based cluster. To Mark Welton from YSU for his expertise and help with configuring the cluster to support our PVM and MPI programs. To Sal Sanders from YSU for providing us with access to Power-PCs running Mac OSX and Adobe Illustrator. To Brian Nelson from YSU for allowing us to test many of our multithreaded and distributed programs on multiprocessor Sun E-250s and E-450s. We are also indebted to Mary Ann Johnson and Jeffrey Trimble from YSU MAAG for helping us locate and hold on to the technical references we required. Claudio M. Stanziola, Paulette Goldweber, and Jacqueline Hansson from the IEEE Standards and Licensing and Contracts Office for obtaining permission to reprint parts of the new Single-UNIX/POSIX standard; Andrew Josey and Gene Pierce from The Open Group was also helpful in this regard. Thanks to Trevor Watkins of the Z-Group for all his help with the testing of the program examples; his multi-Linux distribution environment was especially important in the testing process. A special thanks to Steve Tarasewki for agreeing to provide a technical review for this book while it was in its roughest form. To Dr. Eugene

Santos for pointing us in the right direction as we explored how categorical data structures could be used with PVMs. To Dr. Mike Crescimanno from the Advanced Computing Work Group at YSU for allowing us to present some of the materials from this book at one of the ACWG meetings. Finally, to Paul Petralia and the production team (especially Gail Cocker-Bogusz) from Prentice Hall who had to put up with all of our missed deadlines and strange UNIX/Linux file formats—we are extremely indebted to their patience, encouragement, enthusiasm, and professionalism.

## Chapter 1. The Joys of Concurrent Programming

"I suspect that concurrency is best supported by a library and that such a library can be implemented without major language extensions."

—Bjarne Stroustrup, inventor of C++

In this Chapter

- [What is Concurrency?](#)
- [The Benefits of Parallel Programming](#)
- [The Benefits of Distributed Programming](#)
- [The Minimal Effort Required](#)
- [The Basic Layers of Software Concurrency](#)
- [No Keyword Support for Parallelism in C++](#)
- [Programming Environments for Parallel and Distributed Programming](#)
- [Summary—Toward Concurrency](#)

The software development process now requires a working knowledge of parallel and distributed programming. The requirement for a piece of software to work properly over the Internet, on an intranet, or over some network is almost universal. Once the piece of software is deployed in one or more of these environments it is subjected to the most rigorous of performance demands. The user wants instantaneous and reliable results. In many situations the user wants the software to satisfy many requests at the same time. The capability to perform multiple simultaneous downloads of software and data from the Internet is a typical expectation of the user. Software designed to broadcast video must also be able to render graphics and digitally process sound seamlessly and without interruption. Web server software is often subjected to hundreds of thousands of hits per day. It is not uncommon for frequently used e-mail servers to be forced to survive the stress of a million sent and received messages during business hours. And it's not just the quantity of the messages that can require tremendous work, it's also the content. For instance, data transmissions containing digitized music, movies, or graphics devour network bandwidth and can inflict a serious penalty on server software that has not been properly designed. The typical computing environment is networked and the computers involved have multiple processors. The more the software does, the more it is required to do. To meet the minimal user's requirements, today's software must work harder and smarter. Software must be designed to take advantage of computers that have multiple processors. Since networked computers are more the rule than the exception, software must be designed to correctly and effectively run, with some of its pieces executing simultaneously on different computers. In some cases, the different computers have totally different operating systems with different network protocols! To accommodate these realities, a software development repertoire must include techniques for implementing concurrency through parallel and distributed programming.

## 1.1 What is Concurrency?

Two events are said to be concurrent if they occur within the same time interval. Two or more tasks executing over the same time interval are said to execute concurrently. For our purposes, concurrent doesn't necessarily mean at the same exact instant. For example, two tasks may occur concurrently within the same second but with each task executing within different fractions of the second. The first task may execute for the first tenth of the second and pause, the second task may execute for the next tenth of the second and pause, the first task may start again executing in the third tenth of a second, and so on. Each task may alternate executing. However, the length of a second is so short that it appears that both tasks are executing simultaneously. We may extend this notion to longer time intervals. Two programs performing some task within the same hour continuously make progress of the task during that hour, although they may or may not be executing at the same exact instant. We say that the two programs are executing concurrently for that hour. Tasks that exist at the same time and perform in the same time period are concurrent. Concurrent tasks can execute in a single or multiprocessing environment. In a single processing environment, concurrent tasks exist at the same time and execute within the same time period by context switching. In a multiprocessor environment, if enough processors are free, concurrent tasks may execute at the same instant over the same time period. The determining factor for what makes an acceptable time period for concurrency is relative to the application.

Concurrency techniques are used to allow a computer program to do more work over the same time period or time interval. Rather than designing the program to do one task at a time, the program is broken down in such a way that some of the tasks can be executed concurrently. In some situations, doing more work over the same time period is not the goal. Rather, simplifying the programming solution is the goal. Sometimes it makes more sense to think of the solution to the problem as a set of concurrently executed tasks. For instance, the solution to the problem of losing weight is best thought of as concurrently executed tasks: diet and exercise. That is, the improved diet and exercise regimen are supposed to occur over the same time interval (not necessarily at the same instant). It is typically not very beneficial to do one during one time period and the other within a totally different time period. The concurrency of both processes is the natural form of the solution. Sometimes concurrency is used to make software faster or get done with its work sooner. Sometimes concurrency is used to make software do more work over the same interval where speed is secondary to capacity. For instance, some web sites want customers to stay logged on as long as possible. So it's not how fast they can get the customers on and off of the site that is the concern—it's how many customers the site can support concurrently. So the goal of the software design is to handle as many connections as possible for as long a time period as possible. Finally, concurrency can be used to make the software simpler. Often, one long, complicated sequence of operations can be implemented easier as a series of small, concurrently executing operations. Whether concurrency is used to make the software faster, handle larger loads, or simplify the programming solution, the main object is software improvement using concurrency to make the software better.

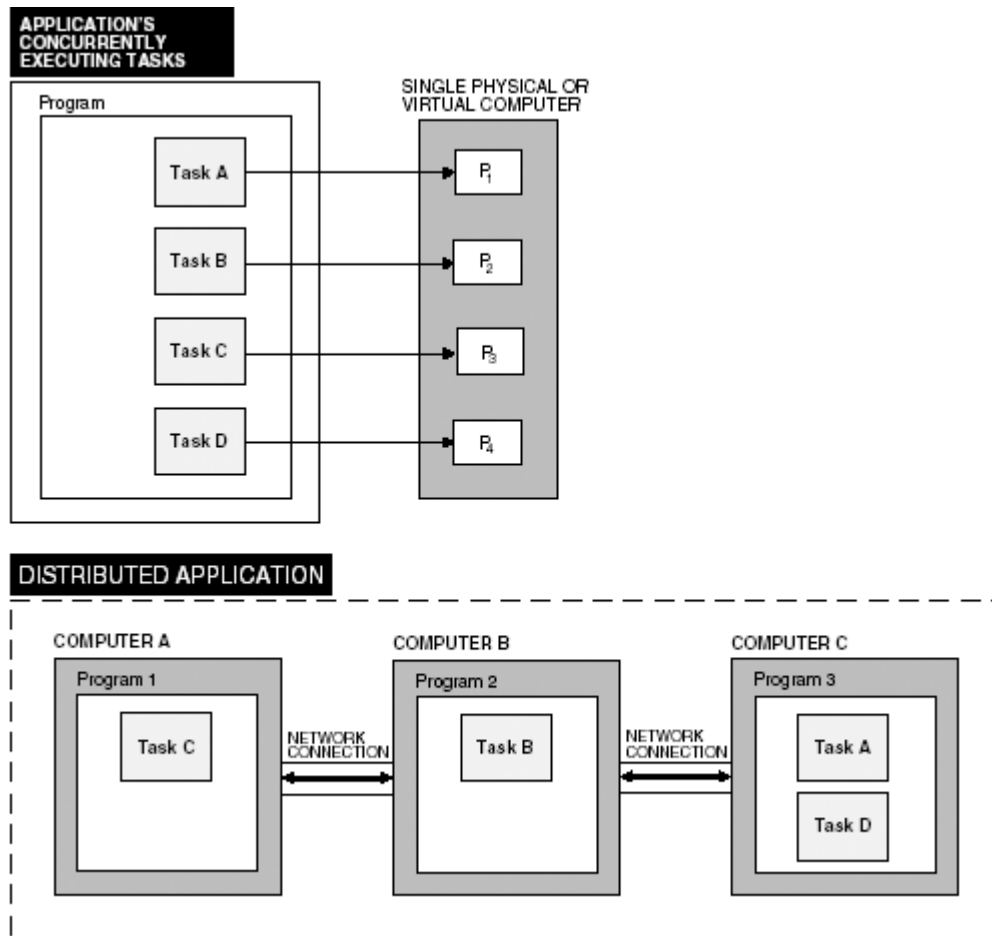
### 1.1.1 The Two Basic Approaches to Achieving Concurrency

Parallel programming and distributed programming are two basic approaches for achieving concurrency with a piece of software. They are two different programming paradigms that sometimes intersect. Parallel programming techniques assign the work a program has to do to two or more processors within a single physical or a single virtual computer. Distributed programming techniques assign the work a program has to do to two or more processes—where the processes may or may not exist on the same computer. That is, the parts of a distributed program often run on different computers connected by a network or at least in different processes. A program that contains parallelism executes on the same physical or virtual computer. The parallelism within a program may be divided into processes or



threads. We discuss processes in [Chapter 3](#) and threads in [Chapter 4](#). For our purposes, distributed programs can only be divided into processes. Multithreading is restricted to parallelism. Technically, parallel programs are sometimes distributed, as is the case with PVM (Parallel Virtual Machine) programming. Distributed programming is sometimes used to implement parallelism, as is the case with MPI (Message Passing Interface) programming. However, not all distributed programs involve parallelism. The parts of a distributed program may execute at different instances and over different time periods. For instance, a software calendar program might be divided into two parts: One part provides the user with a calendar and a method for recording important appointments and the other part provides the user with a set of alarms for each different type of appointment. The user schedules the appointments using part of the software, and the other part of the software executes separately at a different time. The alarms and the scheduling component together make a single application, but they are divided into two separately executing parts. In pure parallelism, the concurrently executing parts are all components of the same program. In distributed programs, the parts are usually implemented as separate programs. [Figure 1-1](#) shows the typical architecture for a parallel and distributed program.

**Figure 1-1. Typical architecture for a parallel and distributed program.**



The parallel application in [Figure 1-1](#) consists of one program divided into four tasks. Each task executes on a separate processor, therefore, each task may execute simultaneously. The tasks can be implemented by either a process or a thread. On the other hand, the distributed application in [Figure 1-1](#) consists of three separate programs with each program executing on a separate computer. Program 3 consists of two separate parts that execute on the same computer. Although Task A and D of Program 3 are on the same computer, they are distributed because they are implemented by two separate processes. Tasks within a parallel program are more tightly coupled than tasks within a distributed program. In general, processors associated with distributed programs are on different computers, whereas processors

associated with programs that involve parallelism are on the same computer. Of course, there are hybrid programs that are both parallel and distributed. These hybrid combinations are becoming the norm.

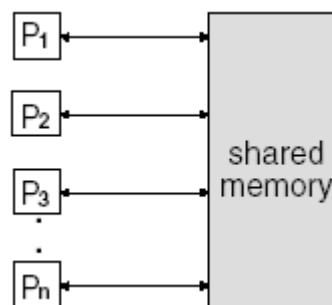
## 1.2 The Benefits of Parallel Programming

Programs that are properly designed to take advantage of parallelism can execute faster than their sequential counterparts, which is a market advantage. In other cases the speed is used to save lives. In these cases faster equates to better. The solutions to certain problems are represented more naturally as a collection of simultaneously executing tasks. This is especially the case in many areas of scientific, mathematical, and artificial intelligence programming. This means that parallel programming techniques can save the software developer work in some situations by allowing the developer to directly implement data structures, algorithms, and heuristics developed by researchers. Specialized hardware can be exploited. For instance, in high-end multimedia programs the logic can be distributed to specialized processors for increased performance, such as specialized graphics chips, digital sound processors, and specialized math processors. These processors can usually be accessed simultaneously. Computers with MPP (Massively Parallel Processors) have hundreds, sometimes thousands of processors and can be used to solve problems that simply cannot realistically be solved using sequential methods. With MPP computers, it's the combination of fast with pure brute force that makes the impossible possible. In this category would fall environmental modeling, space exploration, and several areas in biological research such as the Human Genome Project. Further parallel programming techniques open the door to certain software architectures that are specifically designed for parallel environments. For example, there are certain multiagent and blackboard architectures designed specifically for a parallel processor environment.

### 1.2.1 The Simplest Parallel Model (PRAM)

The easiest method for approaching the basic concepts in parallel programming is through the use of the PRAM (Parallel Random Access Machine). The PRAM is a simplified theoretical model where there are  $n$  processors labeled as  $P_1, P_2, P_3, \dots, P_n$  and each processor shares one global memory. [Figure 1-2](#) shows a simple PRAM.

Figure 1-2. A Simple PRAM.



All the processors have read and write access to a shared global memory. In the PRAM the access can be simultaneous. The assumption is that each processor can perform various arithmetic and logical operations in parallel. Also, each of the theoretical processors in [Figure 1-2](#) can access the global shared memory in one uninterruptible unit of time. The PRAM model has both concurrent and exclusive read algorithms. Concurrent read algorithms are allowed to read the same piece of memory simultaneously with no data corruption. Exclusive read algorithms are used to ensure that no two processors ever read the same memory location at the same time. The PRAM model also has both concurrent and exclusive write algorithms. Concurrent write algorithms allow multiple processors to write to memory, while exclusive write algorithms ensure that no two processors write to the same memory at the same time.

[Table 1-1](#) shows the four basic types of algorithms that can be derived from the read and write possibilities.

**Table 1-1. Four Basic Read-Write Algorithms**

<b>Read-Write Algorithm Type</b>	<b>Meaning</b>
EREW	Exclusive read exclusive write
CREW	Concurrent read exclusive write
ERCW	Exclusive read concurrent write
CRCW	Concurrent read concurrent write

We will refer to these algorithm types often in this book as we discuss methods for implementing concurrent architectures. The blackboard architecture is one of the important architectures that we implement using the PRAM model and it is discussed in [Chapter 13](#). It is important to note that although PRAM is a simplified theoretical model, it is used to develop practical programs, and these programs can compete on performance with programs that were developed using more sophisticated models of parallelism.

### **1.2.2 The Simplest Parallel Classification**

The PRAM gives us a simple model for thinking about how a computer can be divided into processors and memory and gives us some ideas for how those processors may access memory. A simplified scheme for classifying the parallel computers was introduced by M.J. Flynn.[\[1\]](#) These schemes were SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data). These were later extended to SPMD (Single Program Multiple Data) and MPMD (Multiple Program Multiple Data). The SPMD (SIMD) scheme allows multiple processors to execute the same instruction or program with each processor accessing different data. The MPMD (MIMD) scheme allows for multiple processors with each executing different programs or instructions and each with its own data. So in one scheme all the processors execute the same program or instructions and in the other scheme each processor executes different instructions. Of course, there are hybrids of these models where the processors are divided up and some are SPMD and some are MPMD. Using SPMD, all of the processors are simply doing the same thing only with different data. For example, we can divide a single puzzle up into groups and assign each group to a separate processor. Each processor will apply the same rules for trying to put together the puzzle, but each processor has different pieces to work with. When all of the processors are done putting their pieces together, we can see the whole. Using MPMD, each processor executes something different. Even though they are all trying to solve the same problem, they have been assigned a different aspect of the problem. For example, we might divide the work of securing a Web server as a MPMD scheme. Each processor is assigned a different task. For instance, one processor monitors the ports, another processor monitors login attempts, another processor analyzes packet contents, and so on. Each processor works with its own data relative to its area of concern. Although the processors are each doing different work using different data, they are working toward a single solution: security. The parallel programming concepts that we discuss in this book are easily described using PRAM, SPMD (SIMD), and MPMD (MIMD). In fact, these schemes and models are used to implement practical small- to medium-scale applications and should be

sufficient until you are ready to do advanced parallel programming.

[1] M.J. Flynn. Very high-speed computers. In Proceedings of the IEEE, 54, 1901-1909 (December 1966).

## **1.3 The Benefits of Distributed Programming**

Distributed programming techniques allow software to take advantage of resources located on the Internet, on corporate and organization intranets, and on networks. Distributed programming usually involves network programming in one form or another. That is, a program on one computer on a network needs some hardware or software resource that belongs to another computer either on the same network or on some remote network. Distributed programming is all about one program talking to another program over some kind of network connection, which may involve everything from modems to satellites. The distinguishing feature of distributed programs is they are broken into parts. Those parts are usually implemented as separate programs. Those programs typically execute on separate computers and the program's parts communicate with each other over a network. Distributed programming techniques provide access to resources that may be geographically distant. For example, a distributed program divided into a Web server component and a Web client component can execute on two separate computers. The Web server component can be located in Africa and the Web client component can be located in Japan. The Web client part is able to use software and hardware resources of the Web server component, although they are separated by a great distance and almost certainly located on different networks running different operating environments. Distributed programming techniques provide shared access to expensive hardware and software resources. For instance, an expensive, high-end holographic printer may have print server software that provides print services to client software. The print client software resides on one computer and the print server software resides on another computer. Only one print server is needed to serve many print clients. Distributed computing can be used for redundancy and fail over. If we divide the program up into a number of parts with each running on different computers, then we may assign some of the parts the same task. If one of the computers fails for some reason then another part of the same program executing on a different computer picks up the work. Databases can be used to hold billions, trillions, even quadrillions of pieces of information. It is simply not practical for every user to have a copy of the database. The problem is some users are located in different buildings than where the computer with the database is located. Some users are located in different cities, states, and in some instances, countries. Distributed programming techniques are used to allow users to share the massive database regardless of where they are located.

### **1.3.1 The Simplest Distributed Programming Models**

The client-server model of distributed computing is perhaps the easiest to understand and the most commonly used. In this model, a program is divided up into two parts: One part is called the server and the other the client. The server has direct access to some hardware or software resource that the client wants to use. In most cases, the server is located on a different machine than the client. Typically, there is a many-to-one relationship between the server and the client, that is, there is usually one server fulfilling the requests of many clients. The server usually mediates access to a large database, an expensive hardware resource, or an important collection of applications. The client makes requests for data, calculations, and other types of processing. A search engine is a good example of a client-server application. Search engines are used to locate information on the Internet or on corporate and organization intranets. The client is used to obtain a keyword or phrase that the user is interested in. The client software part then passes the request to the server software part. The server has the muscle to perform the massive search for the user's keyword or phrase. The server has either direct access to the information or to other servers that have access to the information. Ideally, the server finds the keyword

or phrase the user requested and returns that information to the client component. Although the client and the server are separate programs on separate computers, they make up a single application. This division of a piece of software into a client and a server is the primary method of distributed programming. The client-server model also has other forms depending on the environment. For instance, the term producer-consumer is a close cousin of client-server. Typically, client-server applications refer to larger programs and producer-consumer refers to smaller programs. Usually when the programs are at the operating system level or lower they are called producer-consumer, and when they are above the operating system level they are usually called client-server (however, there are always exceptions).

### **1.3.2 The Multiagent (Peer-to-Peer) Distributed Model**

Although the client-server model is the most prevalent distributed programming model in use, it is not the only model. Agents are rational software components that are self directed, often autonomous, and can continuously execute. Agents can both make requests of other software components and fulfill requests of other software components. Agents can cooperate within groups to perform certain tasks collectively. In this model there is no specific client or server. The agents form a kind of peer-to-peer model where each of the components are on somewhat equal footing and each component has something to offer to the other. For example, an agent that is providing a price quote for the refurbishing of a vintage sports car might work together with other agents. Where one agent specializes in engine work, another specializes in body work, another specializes in interior design and so on. These agents may cooperatively and collectively come up with the most competitive quote for refurbishing the car. The agents are distributed because each agent is located on a different server on the Internet. The agents use an agreed-upon Internet protocol to communicate. The client-server model is a natural fit for certain types of distributed programming and the peer-to-peer agent model is a natural fit for certain types of distributed programming. We explore both types in this book. The client-server and peer-to-peer models can be used to satisfy most distributed programming demands.

## **1.4 The Minimal Effort Required**

Parallel and distributed programming come with a cost. Although there are many benefits to writing parallel and distributed programming, there are also some challenges and prerequisites. We discuss some challenges in [Chapter 2](#). We mention the prerequisites here. Before a program is written or a piece of software is developed, it must first go through a design process. For parallel and distributed programs, the design process will include three issues: decomposition, communication, and synchronization.

### **1.4.1 Decomposition**

Decomposition is the process of dividing up the problem and the solution into parts. Sometimes the parts are grouped into logical areas (i.e., searching, sorting, calculating, input, output, etc.). In other situations the parts are grouped by logical resource (i.e., file, communication, printer, database, etc.). The decomposition of the software solution amounts to the WBS (work breakdown structure). The WBS determines which piece of software does what. One of the primary issues of concurrent programming is identifying a natural WBS for the software solution at hand. There is no simple or cookbook approach to identifying the WBS. Software development is the process of translating concepts, ideas, patterns of work, rules, algorithms, or formulas into sets of instructions and data that can be executed or manipulated by a computer. This is largely a process of modeling. Software models are reproductions in software of some real-world task, process, or ideal. The purpose of the model is to imitate or duplicate the behavior and characteristics of some real-world entity in a particular domain. This process of modeling uncovers the natural WBS of a software solution. The better the model is

understood and developed the more natural the WBS will be. Our approach is to uncover the parallelism or distribution within a solution through modeling. If parallelism doesn't naturally fit, don't force it. The question of how to break up an application into concurrently executing parts should be answered during the design phase and should be obvious in the model of the solution. If the model of the problem and the solution don't imply or suggest parallelism and distribution then try a sequential solution. If the sequential solution fails, that failure may give clues to how to approach the parallelism.

### **1.4.2 Communication**

Once the software solution is decomposed into a number of concurrently executing parts, those parts will usually do some amount of communicating. How will this communication be performed if the parts are in different processes or different computers? Do the different parts need to share any memory? How will one part of the software know when the other part is done? Which part starts first? How will one component know if another component has failed? These issues have to be considered when designing parallel or distributed systems. If no communication is required between the parts, then the parts don't really constitute a single application.

### **1.4.3 Synchronization**

The WBS designates who does what. When multiple components of software are working on the same problem, they must be coordinated. Some component has to determine when a solution has been reached. The components' order of execution must be coordinated. Do all of the parts start at the same time or does some work while others wait? What two or more components need access to the same resource? Who gets it first? If some of the parts finish their work long before the other parts, should they be assigned new work? Who assigns the new work in such cases? DCS (decomposition, communication, and synchronization) is the minimum that must be considered when approaching parallel or distributed programming. In addition to considering DCS, the location of DCS is also important. There are several layers of concurrency in application development. DCS is applied a little differently in each layer.

## **1.5 The Basic Layers of Software Concurrency**

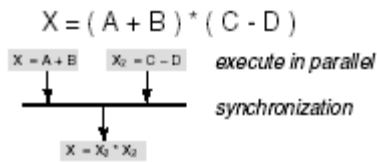
In this book we are concerned with concurrency within the application as opposed to concurrency at the operating system level, or concurrency within hardware. Although the concurrency within hardware and the concurrency at the operating system level support application concurrency, our focus is on the application. For our purposes, concurrency occurs at:

- Instruction level
- Routine (function/procedure) level
- Object level
- Application level

### **1.5.1 Concurrency at the Instruction Level**

Concurrency at the instruction level occurs when multiple parts of a single instruction can be executed simultaneously. [Figure 1-3](#) shows how a single instruction can be decomposed for simultaneous execution.

**Figure 1-3. Decomposition of a single instruction.**



In [Figure 1-3](#), the component  $(A + B)$  can be executed at the same time as  $(C - D)$ . This is an example of concurrency at the instruction level. This kind of parallelism is normally supported by compiler directives and is not under the direct control of a C++ programmer.

### 1.5.2 Concurrency at the Routine Level

The WBS structure of a program may be along function lines, that is, the total work involved in a software solution is divided between a number of functions. If these functions are assigned to threads, then each function can execute on a different processor and if enough processors are available, each function can execute simultaneously. We discuss threads in more detail in [Chapter 4](#).

### 1.5.3 Concurrency at the Object Level

The WBS of a software solution may be distributed between objects. Each object can be assigned to a different thread, or process. Using the CORBA (Common Object Request Broker Architecture) standard, each object may be assigned to a different computer on the network or different computer on a different network. We discuss CORBA in more detail in [Chapter 8](#). Objects residing in different threads or processes may execute their methods concurrently.

### 1.5.4 Concurrency of Applications

Two or more applications can cooperatively work together to solve some problem. Although the application may have originally been designed separately and for different purposes, the principles of code reuse often allow applications to cooperate. In these circumstances two separate applications work together as a single distributed application. For example, the Clipboard was not designed to work with any one application but can be used by a variety of applications on the desktop. Some uses of the Clipboard had not been dreamed of during its original design.

The second and the third layers are the primary layers of concurrency that we will focus on in this book. We show techniques for implementing concurrency in these layers. Operating system and hardware issues are presented only where they are necessary in the context of application design. Once we have an appropriate WBS for a parallel programming or distributed programming design, the question is how do we implement it in C++.

## 1.6 No Keyword Support for Parallelism in C++

The C++ language does not include any keyword primitives for parallelism. The C++ ISO standard is for the most part silent on the topic of multithreading. There is no way within the language to specify that two or more statements should be executed in parallel. Other languages use built-in parallelism as a selling feature. Bjarne Stroustrup, the inventor of the C++ language, had something else in mind. In Stroustrup's opinion:

It is possible to design concurrency support libraries that approach built-in concurrency support both in convenience and efficiency. By relying on libraries, you can support a variety of concurrency models, though, and thus serve the users that need those different models better than can be done by a single built-in concurrency model. I expect this will be

the direction taken by most people and that the portability problems that arise when several concurrency-support libraries are used within the community can be dealt with by a thin layer of interface classes.

Furthermore, Stroustrup says, "I recommend parallelism be represented by libraries within C++ rather than as a general language feature." The authors have found Stroustrup's position and recommendation on parallelism as a library the most practical option. This book is only made possible because of the availability of high-quality libraries that can be used for parallel and distributed programming. The libraries that we use to enhance C++ implement national and international standards for parallelism and distributed programming and are used by thousands of C++ programmers worldwide.

### **1.6.1 The Options for Implementing Parallelism Using C++**

Although there are special versions of C++ that implement parallelism, we present methods on how parallelism can be implemented using the ISO (International Standard Organization) standard for C++. The library approach to parallelism is the most flexible. System libraries and user-level libraries can be used to support parallelism in C++. System libraries are those libraries provided by the operating system environment. For example, the POSIX threads library is a set of system calls that can be used in conjunction with C++ to support parallelism. The POSIX (Portable Operating System Interface) threads are part of the new Single UNIX Specification. The POSIX threads are included in the IEEE Std. 1003.1-2001. The Single UNIX Specification is sponsored by the Open Group and developed by the Austin Common Standards Revision Group. According to the Open Group, the Single UNIX Specification is:

- Designed to give software developers a single set of APIs to be supported by every UNIX system.
- Shifts the focus from incompatible UNIX system product implementations to compliance to a single set of APIs.
- It is the codification and de jure standardization of the common core of UNIX system practice.
- The basic objective is portability of both programmers and application source code.

The Single UNIX Specification Version 3 includes the IEEE Std 1003.1-2001 and the Open Group Base Specifications Issue 6. The IEEE POSIX standards are now a formal part of the Single UNIX Specification and vice versa. There is now a single international standard for a portable operating system interface. C++ developers benefit because this standard contains APIs for creating threads and processes. Excluding instruction-level parallelism, dividing a program up into either threads or processes is the only way to achieve parallelism with C++. The new standard provides the tools to do this. The developer can use:

- POSIX threads (also referred to as pthreads)
- POSIX spawn function
- the exec() family of functions

These are all supported by system API calls and system libraries. If an operation system complies with the Single UNIX Specification Version 3, then these APIs will be available to the C++ developer. These APIs are discussed in [Chapters 3](#) and [4](#). They are used in many of the examples in this book. In addition to system-level libraries, user-level libraries that implement other international standards such as the MPI (Message Passing Interface), PVM (Parallel Virtual Machine), and CORBA (Common Object Request Broker Architecture) can be used to support parallelism with C++.



## 1.6.2 MPI Standard

The MPI is the standard specification for message passing. The MPI was designed for high performance on both massively parallel machines and on workstation clusters. This book uses the MPICH implementation of the MPI standard. MPICH is a freely available, portable implementation of MPI. The MPICH provides the C++ programmer with a set of APIs and libraries that support parallel programming. The MPI is especially useful for SPMD and MPMD programming. The authors use the MPICH implementation of MPI on a 32-node cluster running Linux and an 8-node cluster running Solaris and Linux. Although C++ doesn't have parallel primitives built in, it can take advantage of power libraries such as MPICH that does support parallelism. This is one of the benefits of C++. It is designed for flexibility.

## 1.6.3 PVM: A Standard for Cluster Programming

The PVM is a software package that permits a heterogeneous collection of computers hooked together by a network to be used as a single large parallel computer. The overall objective of the PVM system is to enable a collection of computers to be used cooperatively for concurrent or parallel computation. A PVM library implementation supports:

- Heterogeneity in terms of machines, networks, and applications
- Explicit message-passing model
- Process-based computation
- Multiprocessor support (MPP, SMP)
- Translucent access to hardware (applications can either ignore or take advantage of hardware differences)
- Dynamically configurable host pool (processors can be added and deleted at runtime and can include processor mixes)

The PVM is the easiest to use and most flexible environment available for basic parallel programming tasks that require the involvement of different types of computers running different operating systems. The PVM library is especially useful for several single processor systems that can be networked together to form a virtual parallel processor machine. We discuss techniques for using PVM with C++ in [Chapter 6](#). The PVM is the de facto standard for implementing heterogeneous clusters and is freely available and widely used. The PVM has excellent support for MPMD (MIMD) and SPMD (SIMD) models of parallel programming. The authors use PVM for small- to medium-size parallel programming tasks and the MPI for larger, more complex MPI tasks. PVM and MPI are both libraries that can be used with C++ to do cluster programming.

## 1.6.4 The CORBA Standard

CORBA is the standard for distributed cross-platform object-oriented programming. We mention CORBA here under parallelism because implementations of the CORBA standard can be used to develop multiagent systems. Multiagent systems offer important models of peer-to-peer distributed programming. Multiagent systems can work concurrently. This is one of the areas where parallel programming and distributed programming overlap. Although the agents are executing on different computers, they are executing during the same time period, working cooperatively on a common problem. The CORBA standard provides an open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another

vendor on almost any other computer operating system, programming language, and network. In this book we use the MICO implementation. MICO is a freely available and fully compliant implementation of the CORBA standard. MICO supports C++.

### 1.6.5 Library Implementations Based on Standards

MPICH, PVM, MICO, and POSIX threads are each library implementations based on standards. This means that software developers can rely on these implementations to be widely available and portable across multiple platforms. These libraries are freely available and used by software developers worldwide. The POSIX threads library can be used with C++ to do multithreaded programming. If the program is running on a computer that has multiple processors, then each thread can possibly run on a separate processor and thereby execute concurrently. If only a single processor is available, then the illusion of parallelism is provided and concurrency is achieved through the process of context switching. POSIX threads are perhaps the easiest way to introduce parallelism within a C++ program. Whereas the MPICH, PVM, and MICO libraries will have to be downloaded or obtained (they are readily available), any operating system environment that is client with the POSIX standard or the new UNIX Specification Version 3 will have a POSIX threads implementation. Each library offers a slightly different model of parallelism. [Table 1-2](#) shows how each library can be used with C++.

**Table 1-2. MPICH, PVM, MICO, and POSIX Threads Used with C++**

#### **Libraries C++ Usage**

MPICH	Supports large-scale, complex cluster programming. Strong support for SPMD model. Also supports SMP, MPP, and multiuser configurations.
PVM	Supports cluster programming of heterogeneous environments. Easy to use for single-user, small to medium cluster applications. Also supports MPP.
MICO	Supports either distributed or object-oriented parallel programming. Contains nice support for agent and multiagent programming.
POSIX	Supports parallel processing within a single application at the function or object level. Can be used to take advantage of SMP or MPP.

Whereas languages that depend on built-in support for parallelism are restricted to the models supplied, the C++ developer is free to mix and match parallel programming models. As the nature of the applications change, a C++ developer can select different libraries to match the scenario.

## 1.7 Programming Environments for Parallel and Distributed Programming

The most common environments for parallel and distributed programming are clusters, MPPs, and SMP computers. Clusters are collections of two or more computers that are networked together to provide a single, logical system. The group of computers appear to the application as a single virtual computer. MPP (Massively Parallel Processors) is a single computer that has hundreds of processors. SMP (Symmetric Multiprocessing) is a single system that has processors that are tightly coupled where the processors share memory and the data path. SMP processors share the resources and are all controlled by a single operating system. This book provides a gentle introduction to parallel and distributed

programming, therefore we focus our attention on small clusters of 8 to 32 processors and on multiprocessor machines with 2 to 4 processors. Although many of the techniques we discuss can be used in MPP environments or in large SMP environments, our primary attention is on moderate systems.

## **Summary—Toward Concurrency**

Throughout this book we present an architectural approach to parallel and distributed programming. The emphasis is placed on uncovering the natural parallelism within a problem and its solution. This parallelism is captured within the software model for the solution. We suggest object-oriented methods to help manage the complexity of parallel and distributed programming. Our mantra is function follows form. We use the library approach to provide parallelism support for the C++ language. The libraries we recommend are based on national and international standards. Each library is freely available and widely used. Techniques and concepts presented in the book are vendor independent, nonproprietary, and rely on open standards and open architectures. The C++ programmer and software developer can use different parallel models to serve different needs because each parallelism model is captured within a library. The library approach to parallel and distributed programming gives the C++ programmer the greatest possible flexibility. While parallel and distributed programming can be fun and rewarding, it presents several challenges. In the next chapter we will provide an overview of the most common challenges to parallel and distributed programming.

# Chapter 2. The Challenges of Parallel and Distributed Programming

"The idea that you should really indicate the exact values of any physical quantity — temperature, density, potential field strength or whatever ... is a bold extrapolation."

—Erwin Shrodinger, Causality and Wave Mechanics

In this Chapter

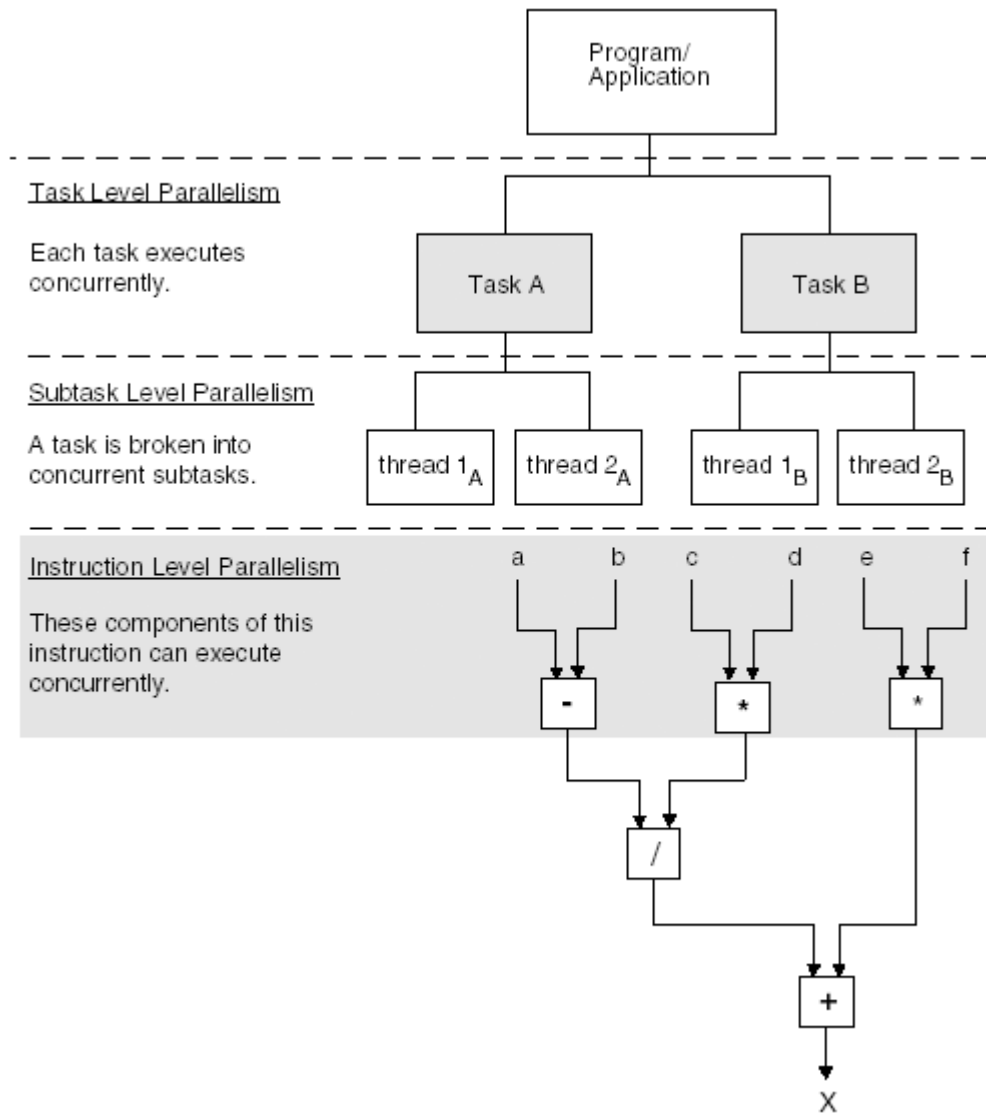
- [The Big Paradigm Shift](#)
- [Coordination Challenges](#)
- [Sometimes Hardware Fails and Software Quits](#)
- [Too Much Parallelization or Distribution Can Have Negative Consequences](#)
- [Selecting a Good Architecture Requires Research](#)
- [Different Techniques for Testing and Debugging are Required](#)
- [The Parallel or Distributed Design Must Be Communicated](#)
- [Summary](#)

In the basic sequential model of programming, a computer program's instructions are executed one at a time. The program is viewed as a recipe and each step is to be performed by the computer in the order and amount specified. The designer of the program breaks up the software into a collection of tasks. Each task is performed in order, and each task must wait its turn. Every program is perceived as having a beginning, middle, and end. The designer or developer envisions each program as a linear progression of tasks. Not only must the tasks march in single file, but the tasks are related so that if the first task cannot complete its work for some reason then the second task may never start. In other words, each task is made to wait on the result of the previous task's work before it can execute. In the sequential model tasks are often serially interdependent. This means that A needs something from B, B needs something from C, C needs something from D, and so on. If B fails for some reason, then C and D will never execute. In a sequential world the developer is accustomed to designing the software to perform step 1 first, then step 2, then step 3. This sequential model is so entrenched in the software design and development process that many programmers find it hard to see things any other way. The solution to every problem, the design of every algorithm, and the layout of every data structure all rely on the computer accessing each instruction or piece of data one at a time.

## 2.1 The Big Paradigm Shift

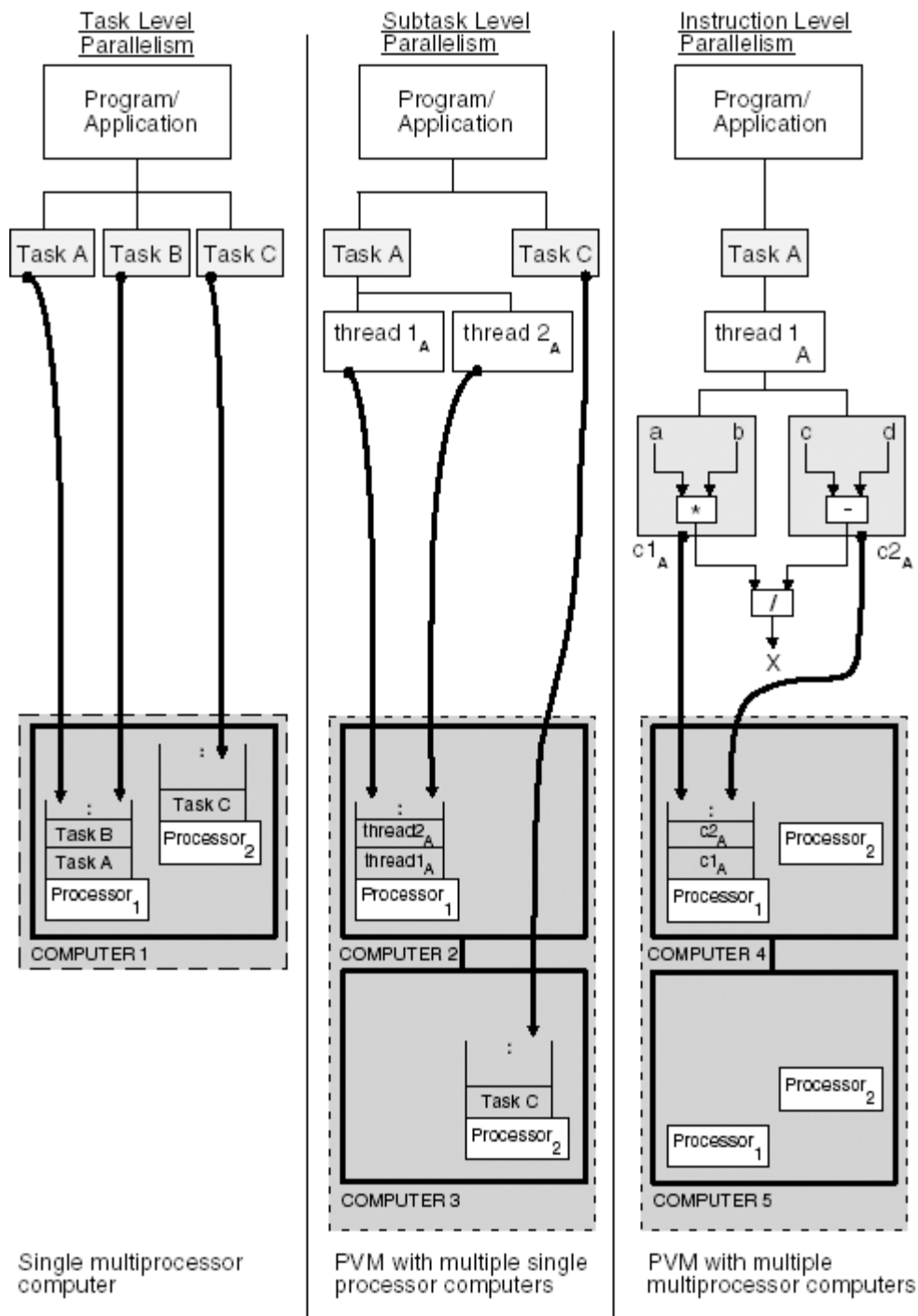
All of this changes in the world of parallel programming. In this world, multiple instructions can execute at the same instant. A single instruction might be broken down into smaller pieces with each piece being executed simultaneously. A program can be broken into multiple tasks that can each execute at the same time. Instead of one task, a program might consist of hundreds or thousands of routines executing concurrently. In the world of parallel programming, the sequence and location of things is not always predictable. Multiple tasks can start at the same time on any processor with no guarantee what task will finish first, in what order they'll finish, or on what processor they will execute. In addition to tasks executing in parallel, a single task may have concurrently executing parts or subtasks. In some configurations, it is possible for the subtasks to run on separate processors, possibly separate computers. [Figure 2-1](#) shows three levels of parallelism that are possible within a single computer program.

**Figure 2-1. The three levels of parallelism that are possible within a single computer program.**



The programmer and developer's model of a program undergoes a big paradigm shift because of the three levels of parallelism shown in [Figure 2-1](#) and how these levels of parallelism can be distributed to multiple processors. [Figure 2-2](#) illustrates how the three levels of parallelism are combined with the basic parallel processor configurations.

Figure 2-2. The three levels of parallelism combined with the basic parallel processor configurations.



Notice in [Figure 2-2](#) that multiple tasks can run on a single processor even when the computer has more than one processor. This situation can be created by operating system scheduling policies. Scheduling policies, process priorities, thread priorities, and input/output device performance all impact where and for how long a task, subtask, or partial instruction will execute. [Figure 2-2](#) emphasizes the different architectures that a programmer must face when moving from the sequential programming model to a parallel programming model. The model changes from a strictly ordered sequence of tasks to only a partially ordered (possibly unordered) collection of tasks. Parallelism turns order of execution, time of execution, and location of execution into wildcards. Any combination of these wildcards is subject to change values in often unpredictable ways.

## 2.2 Coordination Challenges

If a program has routines that can execute in parallel, and these routines share any files, devices, or memory locations, then several coordination issues are introduced. For example, let's say we have an electronic bank withdrawal and deposit program that is divided into three tasks that can execute in parallel. We label the tasks A, B, C.

Task A receives requests from Task B to make withdrawals from an account. Task A also receives requests from Task C to make deposits to an account. Task A accepts and processes requests on a first-come, first-serve basis. What if we have an account that has a balance of \$1,000 and Task C wants to make a \$100 deposit to the account and Task B wants to make a \$1,100 withdrawal from the account? What happens if Task B and Task C both try to update the same account at the same time? What would the balance be? Obviously an account balance can only hold one value at a time. Task A can only apply one transaction at a time to the account, so there's a problem. If Task B's request executes a fraction of a second faster than Task C, then the account will have a negative balance. On the other hand, if Task C gets to the account first, then the account will not have a negative balance. So the balance of the account depends on which task happens to get its request to Task A first. Furthermore, we can execute Tasks B and C several times, each time starting with the same amounts, and sometimes Task B would execute a fraction of a second faster and sometimes Task C would execute faster. Clearly some form of coordination is in order.

To coordinate tasks that are executing in parallel requires communication between the tasks and synchronization of their work. Four common types of problems occur when the communication or the synchronization is incorrect.

### Problem #1 Data Race

If two or more tasks attempt to change a shared piece of data at the same time and the final value of the data depends simply on which tasks get there first, then a race condition has occurred. When two or more tasks are attempting to update the same data resource at the same time, the race condition is called data race. In our electronic banking program, which task gets to the account balance first turns out to be a matter of operating system scheduling, processor states, latency, and chance. This situation creates a race condition. Under these circumstances, what should the bank report as the account's real balance?

So while we would like our electronic banking program to be able to simultaneously handle many banking deposits and withdrawals, we need to coordinate the tasks in the program if the deposits and withdrawals happen to be applied to the same account. Whenever tasks concurrently share a modifiable resource, rules and policies will have to be applied to the task's access. For instance, in our banking program we might apply any deposits to the account before we apply any withdrawals. We might set a rule that only one transaction has access to an account at a time. If more than one transaction for the same account arrives at the same time, then the transactions must be held and organized according to some rule and then granted access one at a time. These organization rules help to accomplish proper synchronization.

### Problem #2 Indefinite Postponement

Scheduling one or more tasks to wait until some event or condition occurs can be tricky. First, the event or condition must take place in a timely fashion. Second, it requires carefully placed communications between tasks. If one or more tasks are waiting for a piece of communication before they execute and that communication either never comes, comes too late, or is incomplete, then the tasks may never execute. Likewise, if the event or condition that we assumed would eventually happen actually never occurs, then the tasks that we have suspended will wait forever. If we suspend one or more tasks waiting on some condition or event that never occurs, this is known as indefinite postponement. In our

electronic banking example, if we set up rules that cause the withdrawal tasks to wait until all deposit tasks are completed, then the withdrawal tasks could be headed for indefinite postponement.

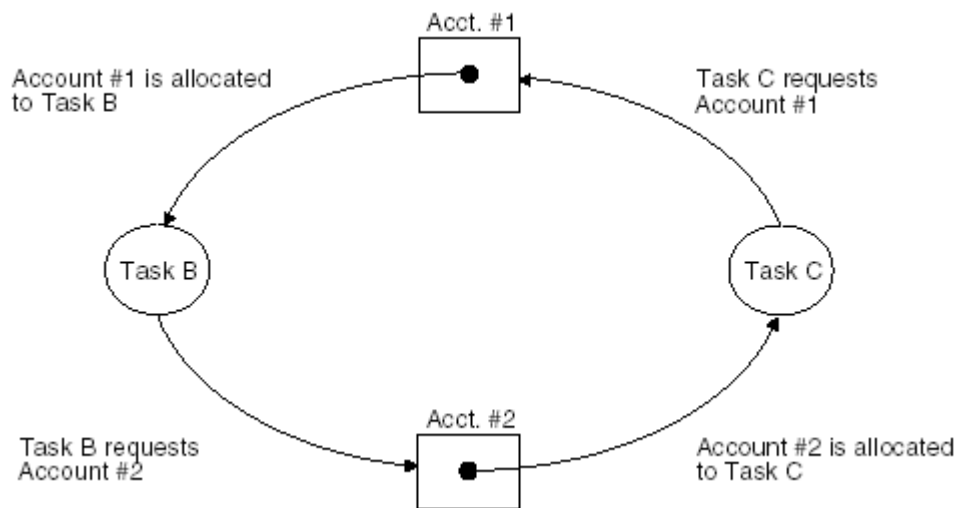
The assumption is that there are deposit tasks. If no deposit requests are made, what will cause the withdrawal tasks to execute? What if the reverse happens, that is, what if deposit requests are continuously made to the same account? As long as a deposit is in progress, no withdrawal can be made. This situation can indefinitely postpone withdrawals.

Indefinite postponement might take place if no deposit tasks appear or if deposit tasks constantly appear. What if deposit requests appear correctly but we fail to properly communicate the event? So as we try to coordinate our parallel task's access to some shared data resource, we have to be mindful of situations that can create indefinite postponement. We discuss techniques for avoiding indefinite postponement in [Chapter 5](#).

### Problem #3 Deadlock

Deadlock is another waiting-type pitfall. To illustrate an example of deadlock, let's assume that the three tasks in our electronic banking program example are working with two accounts instead of one. Recall that Task A receives withdrawal requests from Task B and deposit requests from Task C. Tasks A, B, and C can execute concurrently. However, Tasks B and C may only update one account at a time. Task A grants access on a first-come, first-serve basis. Let's say that Task B has exclusive access to Account 1, and Task C has exclusive access to Account 2. But Task B needs access to Account 2 to complete its processing and Task C needs access to Account 1 to complete its processing. Task B holds on to Account 1 waiting for Task C to release Account 2 and Task C holds on to Account 2 waiting for Task B to release Account 1. Tasks B and C are engaged in a deadly embrace, also known as a deadlock. [Figure 2-3](#) shows the deadlock situation between Tasks B and C.

Figure 2-3. The deadlock situation between Tasks B and C.



The form of deadlock shown in [Figure 2-3](#) requires concurrently executing tasks that have access to some shared writable data to wait on each other for access to that shared data. In [Figure 2-3](#) the shared data are Accounts 1 and 2. Both tasks have access to these accounts. It happens that instead of one task getting access to both accounts at the same time, each task got access to one of the accounts. Since Task B can't release Account 1 until it gets Account 2, and Task C can't release Account 2 until it gets Account 1, the electronic banking program is locked. Notice that Tasks B and C can drive another task(s) into indefinite postponement. If other tasks are waiting on access to Accounts 1 or 2 and Tasks B and C are engaged in a deadlock, then those tasks are waiting for a condition that will never happen. In attempting to coordinate concurrently executing tasks, deadlock and indefinite postponement are two



of the ugliest obstacles that must be overcome.

#### **Problem #4 Communication Difficulties**

Many commonly found parallel environments (e.g., clusters) often consist of heterogeneous computer networks. Heterogeneous computer networks are systems that consist of different types of computers often running different operating systems with different network protocols. The processors involved may have different architectures, different word sizes, and different machine languages. Besides different operating systems, different scheduling and priority schemes might be in effect. To make matters worse, each system might have different qualities of data transmission. This makes error and exception handling particularly challenging. The heterogeneous nature of the system might include other differences. For instance, we might need to share data and logic between programs written in different languages or developed using different software models. The solution might be partially implemented in Fortran, C++, and Java. This introduces interlanguage communication issues. Even when the distributed or parallel environment is not heterogeneous, there is the problem of communicating between two or more processes or between two or more threads. Because each process has its own address space, sharing variables, parameters, and return values between processes requires the use of IPC (inter-process communication) techniques. While IPC is not necessarily difficult, it adds another level of design, testing, and debugging to the system.

The POSIX specification supports five basic mechanisms used to accomplish communication between processes:

- Files with lock and unlock facilities
- Pipes (unnamed and named, also called fifos)
- Shared memory and messaging
- Sockets
- Semaphores

Each of these IPC mechanisms have strengths, weaknesses, traps, and pitfalls that the software designer and developer must manage in order to facilitate reliable and efficient communication between two or more processes. Communication between two or more threads (sometimes called lightweight processes) is easier because threads share a common address space. This means that each thread in the program can easily pass parameters, get return values from functions, and access global data. However, if the communication is not appropriately designed, then deadlock, indefinite postponement, and other data race conditions can easily occur. Both parallel and distributed programming have these four types of coordination problems in common.

Although purely parallel processing systems are different from purely distributed processing systems, we have purposely blurred the lines between the coordination problems in a distributed system versus the coordination problems in a parallel system. This is partly because there is an overlap between the kinds of problems encountered in parallel programming and those encountered in distributed programming. It's also partly because the solutions to the problems in one area are often applicable to problems in the other. However, the primary reason we lose the distinction is that hybrid systems that are part parallel and part distributed are quickly becoming the norm. The state of the art in parallel computing involves clusters, beowolfs, and grids. Exotic cluster configurations is comprised of commodity, off-the-shelf parts that are readily available. These architectures involve multiple computers with multiple processors. Furthermore, single processor systems are on the decline. So, in the future purely distributed systems will be built with computers that have multiple processors (forcing the hybrid issue). This means that as a practical matter the software designer and developer will most likely be faced with problems of distribution and parallelization. Therefore, we discuss the problems in

the same space. [Table 2-1](#) presents a matrix of the combinations of parallel and distributed programming with hardware configurations.

**Table 2-1. A Matrix of the Combinations of Parallel and Distributed Programming with Hardware Configurations**

	<b>Single Computer</b>	<b>Multiple Computers</b>
Parallel programming	Accomplished with multiple processors and breaking up the logic into multiple threads or processes. Threads or processes can run on different processors. IPC required to coordinate tasks.	Accomplished with libraries such as PVM. This requires the type of message passing normally associated with distributed programming.
Distributed programming	Multiple processors are not necessary. The logic may be broken up into multiple processors or threads. IPC required to coordinate tasks.	Accomplished with sockets and components such as CORBA ORB (Object Request Broker). Can use communication that is normally associated with parallel programming.

Notice in [Table 2-1](#) that there are configurations where parallelism is accomplished by using multiple computers. This can be the case using the PVM library. Likewise, there are configurations where distribution can be accomplished on a single computer using multiple processes, or threads. The fact that multiple processes or threads are involved means that the work of the program is "distributed." The combinations in [Table 2-1](#) imply that coordination problems that are normally associated with distributed programming can pop up in parallel programming situations and problems that are normally associated with parallel programming can appear in distributed programming situations.

Regardless of the hardware configuration, there are two basic mechanisms for communicating between two or more tasks: shared memory, and message passing. To effectively use the shared memory mechanism, the programmer must constantly be aware of data race, indefinite postponement, and deadlock pitfalls. The message passing scheme offers other showstoppers such as interrupted transmissions (partial execution), garbled messages, lost messages, wrong messages, too long messages, late messages, early messages, and so on. Much of this book is about the effective use of both mechanisms.

### **2.3 Sometimes Hardware Fails and Software Quits**

When multiple processors are cooperating to provide the solution to some problem, what happens if one or more of the processors fail? Should the program halt or should the work be redistributed somehow? When multiple computers are involved in the solution to some problem, what happens if the communications link between two or more of the computers is temporarily interrupted? What if instead of the communications link being interrupted, the traffic is so slow that processes on each end of the communications time out? How should the software respond in these situations? If we have 50 computers cooperatively solving a problem and only two of the computers fail, should the other 48 pick up the work? If in our electronic banking programming we have a \$1,000 withdrawal and deposit tasks executing simultaneously and two of the tasks are deadlocked, should we shut down the server task? What do we do about the two tasks that are locked? What if the withdrawal and deposit tasks are working properly and for some reason the server task locks up? Should we terminate all the pending withdrawal and deposit tasks? What do we do about partial failures or partial executions? These kinds of considerations are not necessary in single computer sequential programs. Sometimes the failure is a

result of some administration or security policy. For instance, if we have 1,000 routines working on some problem and several of the routines need write access to a file but don't have the write access, this could cause indefinite postponement, deadlock, or partial failure. What if some of the routines are blocked because they don't have security access to the resources they need? Should the entire system be shut down in such cases? How can the information or processing performed be useful if there are hardware interruptions, communications failures, and partial executions? Yet these situations represent normal processing within distributed and parallel environments. In this book, we present several software architectures and programming techniques that can be used to manage these situations.

## **2.4 Too Much Parallelization or Distribution Can Have Negative Consequences**

There is a point where the overhead in managing multiple processors outweighs the speedup and other advantages gained from parallelization. The old adage "you can never have enough processors" is simply not true. Communication between computers or synchronization between processors comes at a cost. The complexity of the synchronization or the amount of communication between processors can require so much computation that the performance of the tasks that are doing the work can be negatively impacted. How many processes, tasks, or threads should a program be divided into? Is there an optimal number of processors for any given parallel program? At what point does adding more processors or computers to the computation pool slow things down instead of speeding them up? It turns out that the numbers change depending on the program. Some scientific simulations may max out at several thousand processors, while for some business applications several hundred might be sufficient. For some client-server configurations, eight processors are optimal and nine processors would cause the server to perform poorly.

There is the work and resources involved in managing parallel hardware and the work involved in managing concurrently executing processes and threads in software. The limit of software processes might be reached before we've reached the optimum number of processors or computers. Likewise, we might see diminishing returns in the hardware before we've reached the optimum number of concurrently executing tasks.

## **2.5 Selecting a Good Architecture Requires Research**

There are many software architectures that support concurrency. The correct architecture needs to be matched with the WBS (Work Breakdown Structure) of a piece of software. Not all parallel and distributed architectures are created equal. While some distributed architectures would work fine in a Web environment, they would fail immediately in a real-time environment. For instance, distributed architectures that support long latency times that would be acceptable in a Web environment are unacceptable for many real-time environments. Compare the distributed processing in a Webbased e-mail system to the distributed processing that takes place with banking ATMs (automated teller machines). Latency that is present in many e-mail systems would simply be unacceptable in real-time systems such as ATMs. Certain distributed architectures (some asynchronous models) manage latency times better than others. Care must also be taken to select the proper parallel processing architectures. For instance, vector processing techniques may work well for certain mathematical and simulation problems, but are ineffective when applied to multiagent planning algorithms. [Table 2-2](#) shows commonly found software architectures that support parallel and distributed programming.

The four basic models listed in [Table 2-2](#) and their variations provide the basic foundations for all the concurrency architectures (i.e., agent, blackboard, object-oriented) that we discuss in this book. It is necessary to become familiar with each of these models and their applications to parallel and distributed programming. We provide an introduction to these models and the bibliography contains material that covers more advanced treatment of each of these models. It is best to find the natural or inherent parallelism in the work being done or in the solution to a problem. The architecture chosen should

match this natural or inherent parallelism as closely as possible. For instance, the parallelism in a solution may be better described using a peer-to-peer model, where all workers are considered equal, as opposed to a boss worker model, where there is a master process managing all the other processes as subordinates.

**Table 2-2. Commonly Found Software Architectures that Support Parallel and Distributed Programming**

<b>Model</b>	<b>Architecture</b>	<b>Distributed Programming</b>	<b>Parallel Programming</b>
Host node also called <ul style="list-style-type: none"> <li>• master/slave</li> <li>• • boss/worker</li> <li>• • loosely (client server)</li> </ul>	Master control tasks that monitors and delegates work to subordinate tasks.	✓	✓
Peer or node only	All tasks are basically equal and work is distributed evenly.		✓
Vector processing loosely pipeline or array processing	One worker for each element of the array or stage in the pipeline.	✓	✓
Tree (parent-child)	Dynamically generated workers in a parent-child relationship. Useful in these types of algorithms: <ul style="list-style-type: none"> <li>• recursion</li> <li>• recursion</li> <li>• AND/OR</li> <li>• tree processing</li> </ul>		

## 2.6 Different Techniques for Testing and Debugging are Required

When testing a sequential program the developer can trace the logic of a program in a step-by-step manner. If the developer starts with the same data and makes sure the system is in the same state, then the outcome or flow of the logic is predictable. The programmer can find bugs in the software by starting the program in the necessary state, using the appropriate input and then tracing through the logic step-by-step. Testing and debugging in the sequential model depends on the predictability of the program's initial state, and current state given the specified input.

This changes with parallel and distributed environments. It is difficult to reproduce the exact context of parallel or distributed tasks because of operating system scheduling policies, dynamic workloads on the computer, processor time slices, process and thread priorities, communication latency, execution latency, and the random chance involved in parallel and distributed contexts. To reproduce the exact state the environment was in during testing and debugging requires that every task the operating system was working on be recreated. The processor scheduling state must be known. The status of virtual memory and context switching all must be reproduced exactly. Interrupt and signal conditions must be recreated. In some cases, networking traffic would have to be recreated! Even the testing and debugging tools impact the exact environment. This means that recreating the same sequence of events in order to test or debug a program is often out of the question. The reason these things would have to be recreated is because they can all help to determine which process or thread can execute and on what processor they can execute. Moreover, it is the particular mix of executing processes and threads that could be the reason for a deadlock, indefinite postponement, data race, or another kind of problem. Although some of these issues also affect sequential programming, they don't disrupt the assumptions of the sequential model. The kind of predictability that is present in the sequential model is simply not available in concurrency programming. This forces the developer to acquire new tactics for testing and debugging programs. It also requires that the developer find new ways to prove program correctness.

## 2.7 The Parallel or Distributed Design Must Be Communicated

There is also the challenge of how to accurately capture a parallel or distributed design in documentation. We must be able to describe the work breakdown structure as well as the synchronization and communication between tasks, objects, processes, and threads. Designers must be able to effectively communicate to developers. Developers must be able to communicate with those that must maintain and administer the system. Ideally, this should be done using a standard notation and representation that is readily available to all concerned. However, finding a single documentation language that is broadly understood and can clearly represent the multiparadigm nature of some of these systems is elusive. We have chosen the UML (Unified Modeling Language) for this purpose. [Table 2-3](#) lists the seven UML diagrams that are helpful for multithreaded, parallel, or distributed programs.

**Table 2-3. Seven UML Diagrams Helpful for Documenting Multithreaded, Parallel, and Distributed Programs**

<b>UML Diagrams</b>	<b>Descriptions</b>
Activity diagram	A type of state diagram in which most (if not all) of the states represent activity and most of the transitions (if not all) are activated by completion of an activity in the source states.
Interaction	A type of diagram that shows the interaction among a set of objects; the interaction is described as a message exchanged among them. These diagrams

## UML Diagrams      Descriptions

include:

- collaboration diagrams
- sequence diagrams
- activity diagrams

State/concurrent state diagram	A diagram that shows the sequence of an object's transformation as it state diagram responds to events. In the case of concurrent state diagram, these transformations can occur during the same time interval.
Sequence diagram	An interaction diagram that shows the organization of the structure of objects that receive and send messages.
Collaboration diagram	An interaction diagram that shows the organization of the structure of objects that receive and send messages.
Deployment diagram	A diagram that shows the runtime configuration of processing nodes, hardware, and software components in a system.
Component diagram	An interaction diagram that shows the dependencies and organization among a set of physical modules of code (packages) in a system.

The seven diagram types in [Table 2-3](#) are only a subset of the diagram types available in the UML, but these types of diagrams are immediately applicable to what we want to capture in our concurrency designs. In particular, the UML's activity, deployment, and state diagrams are very useful in communicating parallel and distributed processing behavior. Since the UML is the de facto standard for communicating object-oriented and agent-oriented designs, we rely upon its use in this book. The Appendix contains a description and explanation for the notation and symbols used in these diagrams.

## Summary

Parallel and distributed programming present challenges in several areas. New approaches to software design and architectures must be adopted. Many of the fundamental assumptions that are held in the sequential model of programming don't apply in the realm of parallel and distributed programming. The four primary coordination problems, data race, indefinite postponement, deadlock, and communication synchronization, are among the major obstacles to programs that require concurrency. Every aspect of the software development life cycle is impacted when the requirements include parallelism or distribution from the initial design down to the testing and documentation. In this book, we present architectural approaches to many of these problems. In addition to the architectural approach, we take advantage of the multiparadigm capabilities of C++ to provide techniques for managing the complexity of parallel and distributed programs.

## Chapter 3. Dividing C++ Programs into Multiple Tasks

"Hence, whatever parallel processes may be going on at a lower (neural) level, at the symbolic level the human mind is fundamentally a serial machine, accomplishing its work through temporal sequences of processes, each typically requiring hundreds of milliseconds for execution."

—Herbert A Simon, *The Machine As Mind*

In this Chapter

- [Process: A Definition](#)
- [Anatomy of a Process](#)
- [Process States](#)
- [Process Scheduling](#)
- [Context Switching](#)
- [Creating a Process](#)
- [Terminating a Process](#)
- [Process Resources](#)
- [What are Asynchronous and Synchronous Processes?](#)
- [Dividing the Program into Tasks](#)
- [Summary](#)

Concurrency in a C++ program is accomplished by factoring your program into either multiple processes or multiple threads. While there are variations on how the logic for a C++ program can be organized (e.g, within objects, functions, generic templates), the options for (with the exception of instruction level) parallelization is accounted for through the use of multiple processes and threads. This chapter focuses on the notion of a process and how C++ programs can be divided into multiple processes.

### 3.1 Process: A Definition

A process is a unit of work created by the operating system. It is important to note that processes and programs are not necessarily equivalent. A program may consist of multiple processes. In some situations, a process might not be associated with any particular program. Processes are artifacts of the operating system and programs are artifacts of the developer. Current operating systems such as UNIX/Linux are capable of managing hundreds or even thousands of concurrently loaded processes.

In order for a unit of work to be called a process it must have an address space assigned to it by the operating system. It must have a process id. It must have a state and an entry in the process table. According to the POSIX standard, it must have one or more flows of controls executing within that address space and the required system resources for those flows of control. A process has a set of executing instructions that reside in the address space of that process. Space is allocated for the instructions, any data that belongs to the process, and stacks for function calls and local variables.

### 3.1.1 Two Kinds of Processes

When a process executes, the operating system assigns the process to a processor. The process executes its instructions for a period a time. The process is preempted so another process can be assigned the processor. The operating system scheduler switches between the code of one process, user, or system to the code of another process, giving each process a chance to execute their instructions. There are user and system processes. Processes that execute system code are called system processes. System processes administer to the whole system. They perform housekeeping tasks such as allocating memory, swapping pages of memory between internal and secondary storage, checking devices, and so on. They also perform tasks on behalf of the user processes such as fulfill I/O requests, allocate memory, and so on. User processes execute its own code and sometimes they make system function calls. When a user process executes its own code, it is in user mode. In user mode, the process cannot execute certain privileged machine instructions. When a user process makes a system function call, for example read(), write(), open(), it is executing operating system instructions. What occurs is the user process is put on hold until the system call has completed. The processor is given to the kernel to complete the system call. At that time the user process is said to be in kernel mode and cannot be preempted by any user processes.

### 3.1.2 Process Control Block

Processes have characteristics used for identification and determining their behavior during execution. The kernel maintains data structures and provides system functions that allow the user to have access to this information. Some information is stored in the PCB (process control block). The information stored in the PCB describes the process to the operating system. This information is needed in order for the operating system to manage each process. When the operating system switches between a process utilizing the CPU to another process, it saves the current state of the executing process and its context to the PCB save area in order to restart the process the next time it is assigned to the CPU. The PCB is read and changed by various modules of the operating system. Modules concerned with the monitoring the operating system's performance, scheduling, allocating resources, and interrupt processing access and/or modify the PCB. PCB information includes:

- current state and priority of the process
- process, parent, and child identifiers
- pointers to allocated resources
- pointers to location of the process's memory
- pointer to the process's parent and child processes
- processor utilized by process
- control and status registers
- stack pointers

The information stored in the PCB can be organized as information concerned with process control such as the current state and priority of the process, pointers to parent/child PCBs, allocated resources, and memory. This also includes any scheduling-related information, process privileges, flags, messages, and signals that have to do with communication between processes (IPC, or interprocess communication). The process control information is required by the operating system in order to coordinate the concurrently active processes. Stack pointers and the content of user, control, and status registers describe information concerned with the state of the processor. When a process is running, information is placed in the registers of the CPU. Once the operating system decides to switch to

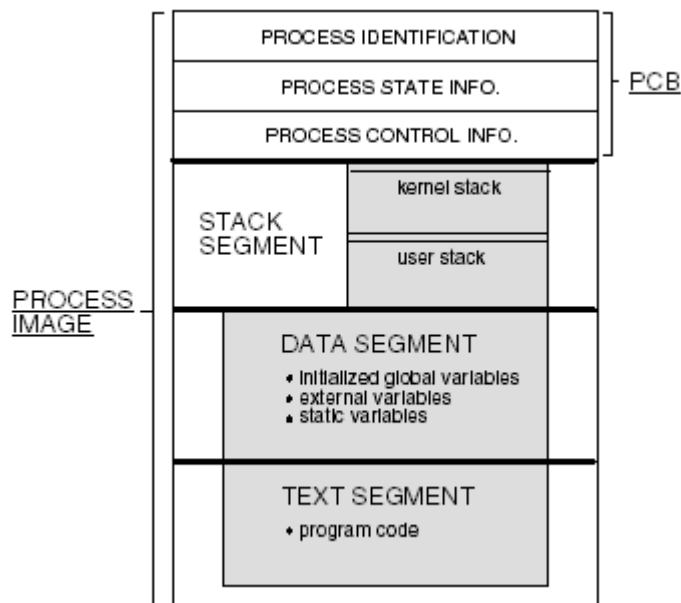


another process, all the information in those registers has to be saved. When the process gains the use of the CPU again, this information can be restored. Other information has to do with process identification. This is the process id (PID), and the parent id (PPID). These identification numbers are unique for each process. They are positive, nonzero integers.

### 3.2 Anatomy of a Process

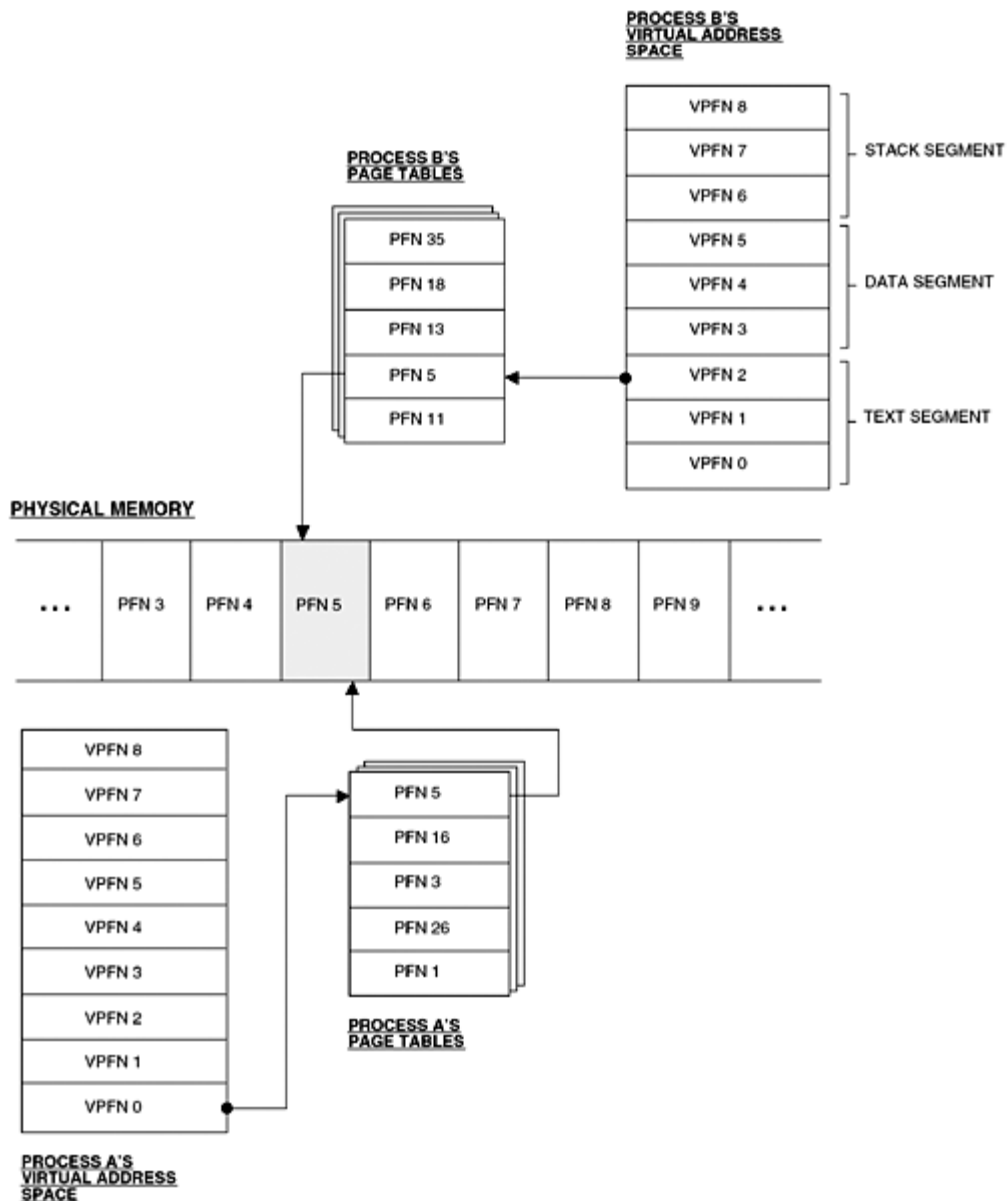
The address space of a process is divided into three logical segments: text (program code), data and stack segments. [Figure 3-1](#) shows the logical layout of a process. The text segment is at the bottom of the address space. The text segment contains the instructions to be executed, called the program code. The data segment above it contains the initialized global, external, and static variables for the process. The stack segment contains locally allocated variables and parameters passed to functions. Because a process can make system function calls as well as user-defined function calls, two stacks are maintained in the stack segment, the user-stack and the kernel-stack. When a function call is made, a stack-frame is constructed and pushed onto either the user or kernel stack depending on whether the process is in user or kernel mode. The stack segment grows downward toward the data segment. The stack frame is popped from the stack when the function returns. The text, data, stack segments, and process control block are part of what forms the process image.

**Figure 3-1. The address space of a process divided into the text, data, and stack segments. This is the logical layout of a process.**



The address space of a process is virtual. Virtual storage dissociates the addresses referenced in an executing process from the addresses actually available in internal memory. This allows the addressing of storage space much larger than what is available. The segments of the process's virtual address space are contiguous blocks of memory. Each segment and physical address space are broken up into chunks called pages. Each page has a unique page frame number. The virtual page frame number is used as an index into the process's page tables. The page table entries contain a physical page frame number, thus mapping the virtual page frames to physical page frames. This is depicted in [Figure 3-2](#). As illustrated, virtual address space is contiguous but it is mapped to physical pages in any order.

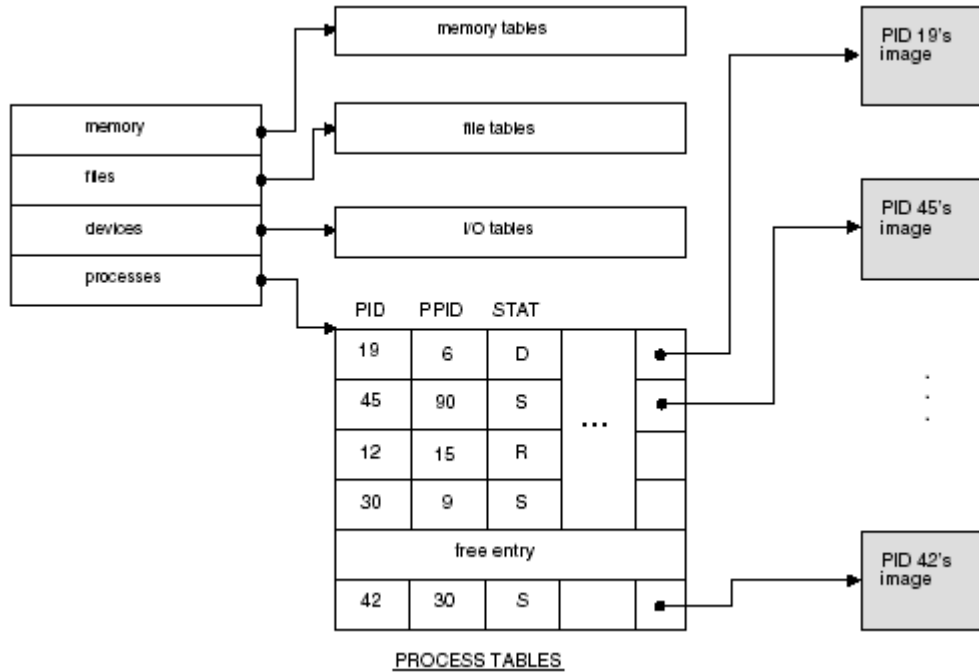
Figure 3-2. The contiguous virtual page frames mapped to pages in physical memory.



Although the virtual address space of each process is protected to prevent another process from accessing it, the text segment of a process can be shared among several processes. [Figure 3-2](#) also shows how two processes can share the same program code. The same physical page frame number is stored in the page table entries of both processes' page tables. As illustrated in [Figure 3-2](#), process A virtual page frame 0 is mapped to physical page frame 5 as well as process B's virtual page frame 2.

In order for the operating system to manage all the processes stored in internal memory, it creates and maintains process tables. Actually, the operating system has a table for all of the entities that it manages. Keep in mind that the operating system manages not only processes but all the resources of the computer including devices, memory, and files. Some of the memory, devices, and files are managed on behalf of the user processes. This information is referenced in the PCB as resources allocated to the process. The process table will have an entry for each process image in memory. Each entry contains the process and parent process id, real and effective user id and group id, list of pending signals, the location of the text, data, and stack segments, and the current state of the process. When the operating system needs to access a process, the process is looked up in the process table and then the process image is located in memory. This is depicted in [Figure 3-3](#).

Figure 3-3. The operating system control tables. Each entry in the process table stores represents a process in the system.



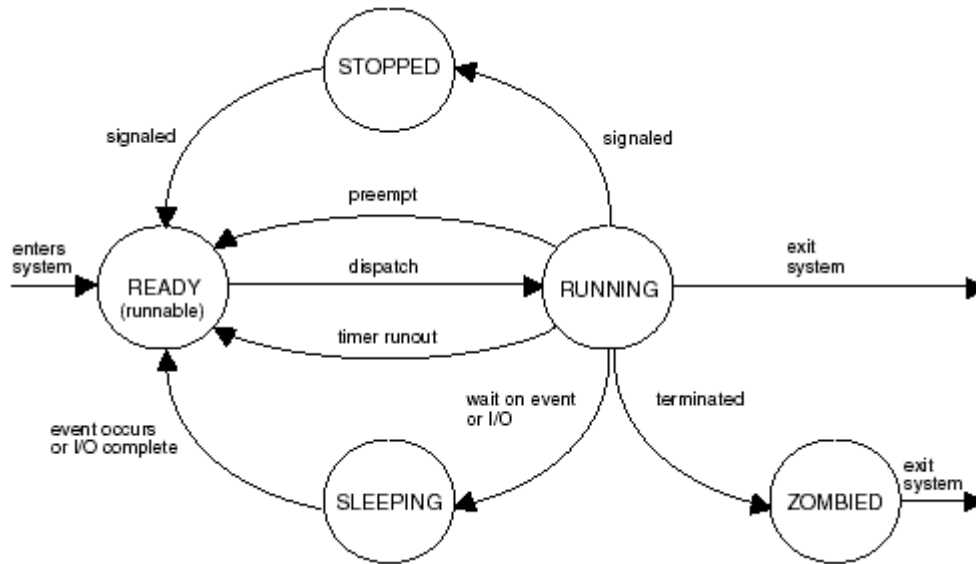
### 3.3 Process States

During a process' execution, the process's state changes. The state of the process is the current condition or status of the process. In the UNIX environment, a process can be in the following states:

- running
- runnable (ready)
- zombied
- waiting (blocked)
- stopped

The process changes its state when certain circumstances created by the process or the operating system exist. The state transition is the circumstance that causes the process to change its state. [Figure 3-4](#) is the state diagram for the UNIX environment. The state diagram has nodes and directed edges between the nodes. Each node represents the state of the process. The directed edges between the nodes are state transitions. [Table 3-1](#) lists the state transitions with a brief description. As [Figure 3-4](#) and [Table 3-1](#) show, only certain transitions are allowed between states. For example, there is a transition, an edge, between ready and running but there is no transition between sleeping and running, meaning there are circumstances that cause a process to move from the ready state to the running state but there are no circumstances that cause a process to move from the sleeping state to a running state.

Figure 3-4. The process states and transitions in the UNIX/Linux environments.



When a process is created, it is ready to execute its instructions but must first wait until the processor is available. Each process is only allowed to use a processor for a discrete interval called a time slice. Processes waiting to use a processor are placed in a ready queues. Only processes in the ready queues are selected (by the scheduler) to use the processor. Processes in the ready queues are runnable. When the processor is available, a runnable process is assigned a processor by the dispatcher. When the time slice has expired, the process is removed from the processor, whether it has finished executing all its instructions or not. The process is placed back in the ready queue to wait for its next turn to use the processor. A new process is selected from a ready queue and is given its time slice to execute. System processes are not preempted. When they are given the processor, they run until completion. If the time slice has not expired, a process may voluntarily give up the processor if it cannot continue to execute. The process may have made a request to access an I/O device by making a system call or it may need to wait on a synchronization variable to be released. Processes that cannot continue to execute because they are waiting for an event to occur are in a sleeping state. They are placed in a queue with other sleeping processes. They are removed from that queue and placed back in the ready queue when the event occurs. The processor may be taken away from a process before its time slice has run out if a process with a higher priority, like a system process, is runnable. The preempted process is still runnable and therefore placed back in the ready queue.

Table 3-1. Process Transitions

State transitions	Descriptions
READY → RUNNING (dispatch)	The process is assigned to the processor.
RUNNING → READY (timer runout)	The time slice the process is assigned to the processor has run out. The process is placed back in the ready queue.
RUNNING → READY (preempt)	The process has been preempted before the time slice ran out. This can occur if a process with a higher priority is runnable. The process is placed back in

State transitions	Descriptions
	the ready queue.
RUNNING → SLEEPING (block)	The process gives up the processor before the time slice has run out. The process may need to wait for an event or has made a system call, for example, a request for I/O. The process is placed in a queue with other sleeping processes.
SLEEPING → READY (unblock)	The event the process was waiting for has occurred or the system call has completed, for example, I/O request is filled. The process is placed back in the ready queue.
RUNNING → STOPPED	The process gives up the processor because it has received a signal to stop.
STOPPED → READY	The process has received the signal to continue and is placed back in the ready queue.
RUNNING → ZOMBIED	The process has been terminated and awaits the parent to retrieve its exit status from the process table.
ZOMBIED → EXIT	The parent process has retrieved the exit status and the process exits the system.
RUNNING → EXIT	The process has terminated, the parent has retrieved the exit status, and the process exits the system.

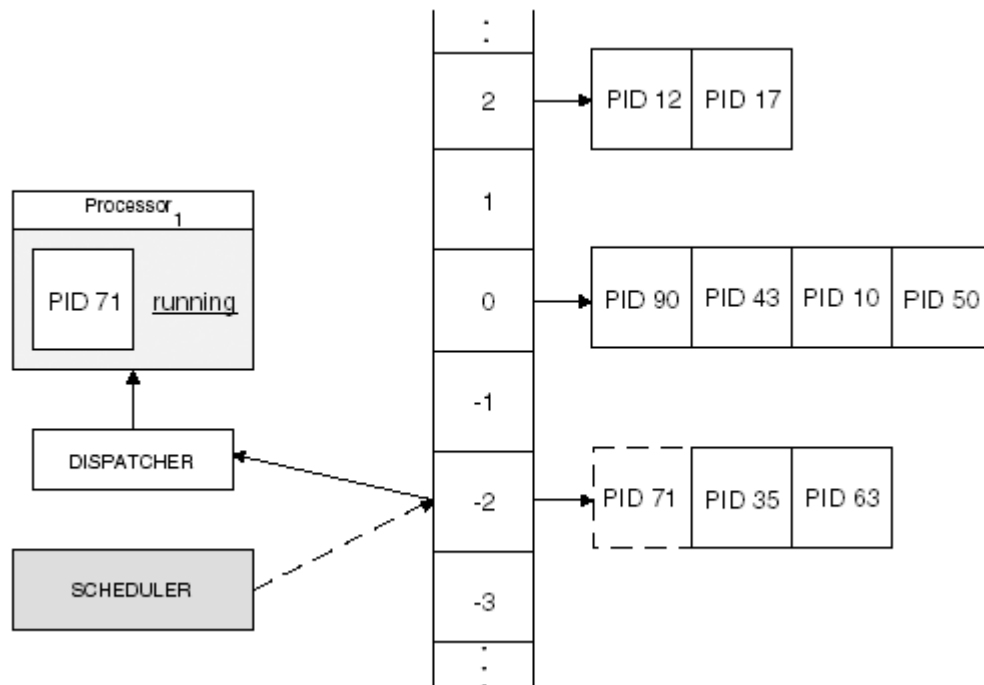
A running process can receive a signal to stop executing. The stopped state is different from a sleeping state because the time slice has not expired nor has the process made any requests of the system. The process may receive a signal to stop because it is being debugged or some situation in the system has occurred. The process makes a transition from running state to stopped state. Later, the process may be awakened or destroyed.

When a process has executed all its instructions, it exits the system. The process is removed from the process table, the PCB is destroyed, and all of its resources are deallocated and returned to the system pool of available resources. A process that is unable to continue executing and cannot exit the system is zombied. A zombied process does not use any system resources but it still maintains an entry in the process table. When the process tables contain too many zombied processes, the performance of the system is affected, which can possibly cause the system to reboot.

### 3.4 Process Scheduling

When a ready queue contains several processes, the scheduler must determine which process should be assigned to a processor first. The scheduler maintains data structures that allow it to schedule the processes in an efficient manner. Each process is given a priority class and placed in a priority queue with other runnable processes with the same priority class. There are multiple priority queues, each representing a different priority class used by the system. These priority queues are stratified and placed in a dispatch array called the multilevel priority queue, illustrated in [Figure 3-5](#). Each element in the array points to a priority queue. The scheduler assigns the process at the head of the nonempty highest priority queue to the processor.

**Figure 3-5. The multilevel priority queue in which each entry of the dispatch array points to a ready queue of processes with the same priority level.**



Priorities can be dynamic or static. Once a static priority of a process is set, it cannot be changed. Dynamic priorities can be changed. Processes of the highest priority can monopolize the use of the processor. If the priority of a process is dynamic, the initial priority can be adjusted to a more appropriate value. The process is placed in a priority queue that has a higher priority. A process monopolizing the processor can also be given a lower priority or other processes can be given a higher priority than that process. In the UNIX/Linux environments, the range of priority levels is from -20 to 19. The higher the value, the lower the priority.

When assigning priority to a user process, what the process spends most of its time doing should be considered. Some processes are CPU intensive. CPU-intensive processes use the processor for the whole time slice. Some processes spend most of their time waiting for I/O or some other event to occur. When such a process is ready to use a processor, it should be given the processor immediately so it can make its next request for I/O. Processes that are interactive may require a high priority to ensure good response time. System processes have a higher priority than user processes.

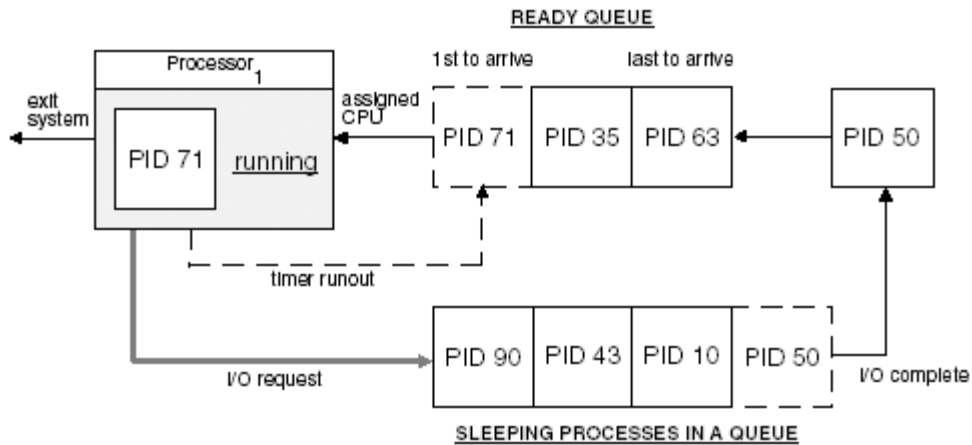
#### 3.4.1 Scheduling Policy

The processes are placed in a priority queue according to a scheduling policy. Two of the scheduling

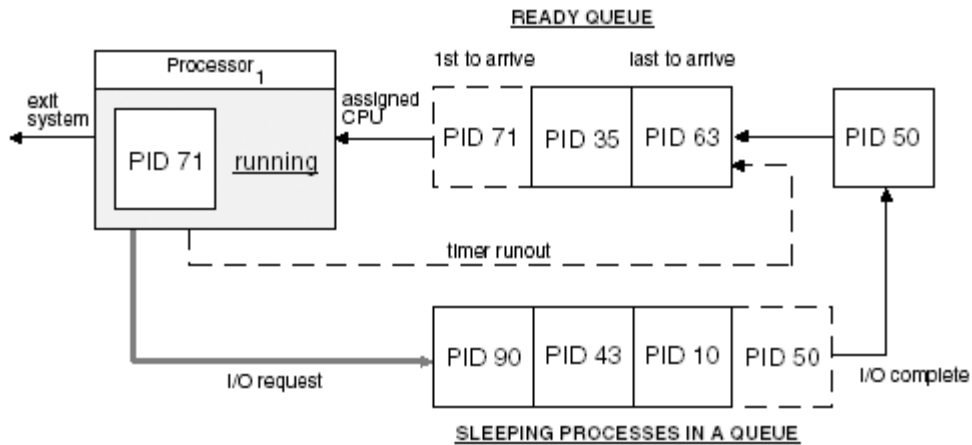
policies used by the UNIX/Linux systems are FIFO (First-In-First-Out) and round-robin (RR) policies. [Figure 3-6\(a\)](#) shows the FIFO scheduling policy. With a FIFO scheduling policy, processes are assigned a processor according to the arrival time in the queue. When a running process time slice has expired, it is placed at the head of its priority queue. When a sleeping process becomes runnable, the process is placed at the end of its priority queue. A process can make a system call and give up a processor to another process with the same priority level. The process will be placed at the end of its priority queue.

**Figure 3-6. The behavior of the First-In-First-Out (FIFO) and round-robin (RR) scheduling policies. The FIFO scheduling policy assigns processes to the processor according to its arrival time in the queue. The process runs until completion. The RR scheduling policy assigns processes using FIFO scheduling but when the time slice runs out the process is placed at the back of the ready queue.**

(a) FIFO SCHEDULING



(b) RR SCHEDULING



In a round-robin scheduling policy, all processes are considered equal. [Figure 3-6\(b\)](#) depicts the RR scheduling policy. RR scheduling is the same as FIFO scheduling with an exception: when the time slice expires, the process is placed at the back of the queue and the next process in the queue is assigned the processor.

### 3.4.2 Using the ps Utility

The ps utility generates a report that summarizes execution statistics for the current processes. This information can be used to monitor the status of current processes. [Table 3-2](#) lists the common headers

and the meaning of the output for the ps utility for the Solaris/Linux environments. In a multi-processor environment, the ps utility is quite useful to monitor the state, CPU and memory usage, processor utilized, priority, and start time of the current processes executing. Command options control which processes are listed and what information is displayed about each process. In the Solaris environment, by default (no command options used), information about processes with the same effective user id and controlling terminal of the calling invoker is displayed. In the Linux environment, by default, the processes with the same user id as the invoker are displayed. In both environments, the only information that will be displayed is PID, TTY, TIME and COMMAND. These are some of the options that control which processes are displayed:

- t List the processes associated with the terminal specified by term  
term
- e All current processes
- a (Linux) All processes with tty terminal except the session leaders  
  
(Solaris) Most frequently requested processes except group leaders and processes not associated with a terminal
- d All current processes except session leaders
- T (Linux) All processes in this terminal
- a (Linux) All processes including those of other users
- r (Linux) Only running processes

**Table 3-2. Common Headers Used for ps Utility in the Solaris/Linux Environments**

<b>Headers</b>	<b>Description</b>
USER, UID	Username of process owner
PID	Process ID
PPID	Parent process ID
PGID	ID of process group leader
SID	ID of session leader



<b>Headers</b>	<b>Description</b>
%CPU	Percentage of CPU time used by the process in the last minute
RSS	Amount of real RAM currently used by the process in k
%MEM	Percentage of real RAM used by the process in the last minute
SZ	Size of virtual memory of the process's data and stack in k or pages
WCHAN	Address of an event for which a process is sleeping
COMMAND CMD	Command name and arguments
TT, TTY	Process's controlling terminal
S, STAT	Current state of the process
TIME	Total CPU time used by the process (HH:MM:SS)
STIME, START	Time or date the process started
NI	Nice value of the process
PRI	Priority of the process
C, CP	Short-term CPU-use factor used by the scheduler to compute PRI
ADDR	Memory address of a process
LWP	ID of the lwp (thread)
NLWP	The number of lwps

## Synopsis

```
(Linux)
ps -[Unix98 options]
  [BSD-style options]
 --[GNU-style long options]
```

```
(Solaris)
ps [-aAdeflcljLPy] [-o format] [-t termlist][--u userlist]
    [-G grouplist][--p proclist] [-g pgrplist] [-s sidlist]
```

The following list contains some of the command options used to control the information displayed about the processes:

-f                    full listings

-l                    long format

-j                    jobs format

Below is an example of using the ps utility in Solaris/Linux environments:

```
ps -f
```

This will display information about the default processes in each environment. [Figure 3-7](#) shows the output in the Solaris environment. The command options can also be used in tandem. [Figure 3-7](#) also shows the output of using -l and -f together in the Solaris environment:

```
ps -lf
```

**Figure 3-7 Output of ps -f and ps -lf in the Solaris environment.**

```
//SOLARIS
$ ps -f
  UID  PID  PPID  C   STIME   TTY  TIME CMD
cameron 2214 2212  0 21:03:35 pts/12 0:00 -ksh
cameron 2396 2214  2 11:55:49 pts/12 0:01 nedit

$ ps -lf
F S   UID  PID  PPID  C  PRI NI   ADDR  SZ   WCHAN   STIME  TTY  TIME  CMD
8 S  cameron 2214 2212  0   51 20 70e80f00 230 70e80f6c 21:03:35 pts/12 0:00 -ksh
8 S  cameron 2396 2214  1   53 24 70d747b8 843 70152aba 11:55:49 pts/12 0:01 nedit
```

The l command option shows the additional headers: F, S, C, PRI, NI, ADDR, and WCHAN. The P command option will display the PSR header. Under this header is the number of the processor to which the process is assigned or bound.

[Figure 3-8](#) shows the output of the ps utility using the Tux command options in the Linux environment. The %CPU, %MEM, and STAT information is displayed for the processes. In a multiprocessor environment, this information can be used to monitor which processes are dominating CPU and memory usage. The STAT header shows the state or status of the process. [Table 3-3](#) lists how the status is encoded and their meanings. The STAT header can reveal additional information about the status of the process:

D        (BSD) Disk wait

P        (BSD) Page wait

- X (System V) Growing: waiting for memory
- W (BSD) Swapped out
- K (AIX) Available kernel process
- N (BSD) Niced: execution priority lowered
- > (BSD) Niced: execution priority artificially raised
- < (Linux) High priority process
- L (Linux) Pages are locked in memory

**Figure 3-8 Output of ps Tux in the Linux environment.**

//Linux

```
[tdhughes@colony]$ ps Tux
USER      PID %CPU %MEM  VSZ  RSS  TTY  STAT  START  TIME  COMMAND
tdhughes 19259  0.0  0.1  2448 1356 pts/4   S   20:29  0:00  -bash
tdhughes 19334  0.0  0.0  1732  860 pts/4   S   20:33  0:00  /home/tdhughes/pv
tdhughes 19336  0.0  0.0  1928  780 pts/4   S   20:33  0:00  /home/tdhughes/pv
tdhughes 19337 18.0  2.4 26872 24856 pts/4   R   20:33  0:47  /home/tdhughes/pv
tdhughes 19338 18.0  2.3 26872 24696 pts/4   R   20:33  0:47  /home/tdhughes/pv
tdhughes 19341 17.9  2.3 26872 24556 pts/4   R   20:33  0:47  /home/tdhughes/pv
tdhughes 19400  0.0  0.0  2544  692 pts/4   R   20:38  0:00  ps Tux
tdhughes 19401  0.0  0.1  2448 1356 pts/4   R   20:38  0:00  -bash
```

These codes will precede the status codes. If an N precedes the status, this means that the process is running at a lower priority level. If a process has a status SW<, this means the process is sleep, swapped out, and has a high priority level.

### 3.4.3 Setting and Returning the Process Priority

The priority level of a process can be changed by using the nice() function. Each process has a nice value that is used to calculate the priority level of the calling process. A process inherits the priority of the process that created it. The priority of a process can be lowered by raising its nice value. Only superuser and kernel processes can raise their priority levels.

**Synopsis**

```
#include <unistd.h>
int nice(int incr);
```

A low nice value raises the priority level of the process. The `incr` parameter is the value added to the current nice value of the calling process. The `incr` can be negative or positive. The nice value is a non-negative number. A positive `incr` value will raise the nice value, therefore lowering the priority level. A negative `incr` value will lower the nice value, therefore raising the priority level. If the `incr` value raises the nice value above or below its limits, the nice value of the process will be set to the highest or lowest limit accordingly. If successful, the `nice()` function will return the new nice value of the process. If unsuccessful, the function will return -1 and the nice value is not changed.

## Synopsis

```
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int value);
```

The `setpriority()` function sets the nice value for a process, process group, or user. The `getpriority()` returns the priority of a process, process group, or user. [Example 3.1](#) shows the syntax to the functions `setpriority()` and `getpriority()` to set and return the nice value of the current process.

### Example 3.1 Using `setpriority()` and `getpriority()`.

```
#include <sys/resource.h>

//...
id_t pid = 0;
int which = PRIO_PROCESS;
int value = 10;
int nice_value;
int ret;
nice_value = getpriority(which,pid);
if(nice_value < value) {
    ret = setpriority(which,pid,value);
}
//...
```

In [Example 3.1](#), the priority of the calling process is being returned and set. If the calling process's nice value is < 10, the nice value of the process is set to 10. The target process is determined by the values stored in the `which` and `who` parameters. The `which` parameter can specify a process, process group, or user. It can have the following values:

<code>PRIO_PROCESS</code>	Indicates a process
<code>PRIO_PGRP</code>	Indicates a process group
<code>PRIO_USER</code>	Indicates a user

Depending on the value of `which`, the `who` parameter is the id number of a process, process group, or effective user. In [Example 3.1](#), `which` is assigned `PRIO_PROCESS`. A 0 value for `who` indicates the current process, process group, or user. In [Example 3.1](#), the `who` is set to 0, indicating the current process. The value parameter for `setpriority()` shall be the new nice value for the specified process, process group, or user. The range of nice value in the Linux environment is -20 to 19. In [Example 3.1](#),

the value of nice is set to 10 if the current nice value is less than 10. Unlike the function nice(), the value passed to setpriority() is the actual value of nice and not an offset to be added to the current nice value.

In a process with multiple threads, the modification of the priority will affect the priority of all the threads in that process. If successful, getpriority() will return the nice value of the specified process. If successful, setpriority() will return 0. If unsuccessful, both functions will return -1. The return value -1 is a legitimate nice value for a process. To determine if an error has occurred, check the external variable errno.

### **3.5 Context Switching**

A context switch occurs when the use of the processor is switched from one process to another process. When a context switch occurs, the system saves the context of the current running process and restores the context of the next process selected to use the processor. The PCB of the preempted process is updated. The process state field is changed from the running to the appropriate state (runnable, blocked, zombied, etc.). The contents of the processor's registers, state of the stack, user and process identification and privileges, and scheduling and accounting information are saved and updated.

The system must keep track of the status of the process's I/O and other resources, and any memory management data structures. The preempted process is placed in the appropriate queue.

A context switch occurs when:

- a process is preempted
- a process voluntarily gives up the processor
- a process makes an I/O request or needs to wait for an event
- a process switches from user mode to kernel mode

When the preempted process is selected again to use the processor, its context is restored and execution continues where it left off.

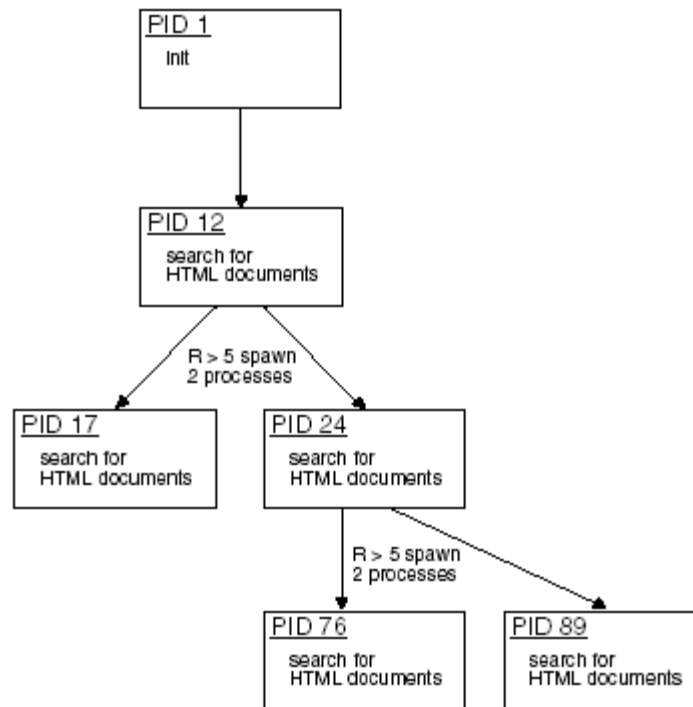
## 3.6 Creating a Process

To run any program the operating system must first create a process. When a new process is created, a new entry is placed in the main process table. A new PCB is created and initialized and the process identification portion of the PCB contains a unique process id number and the parent process id. The program counter is set to point to the program entry point and the system stack pointers are set to define the stack boundaries for the process. The process is initialized with any of the attributes requested. If the process is not given a priority value, it is given the lowest priority value by default. The process initially does not own any resources unless there is an explicit request for resources or they have been inherited from the creator process. The state of the process is runnable and placed in the runnable or ready queue. Address space is allocated for the process. How much space to be set aside can be determined by default based on the type of process. The size can also be set as a request by the creator of the process. The creator process can pass the size of the address space to the system at the time the process is created.

### 3.6.1 Parent–Child Process Relationship

A process that creates or spawns another process is a parent process to the spawned child process. The init process is the parent of all user processes. The init process is the very first process visible to the UNIX system when booted up. The init process brings the system up, runs other programs when necessary, and starts daemons. It has a PID of 1. The child process has its own unique PID, PCB, and a separate entry in the process table. The child process can also spawn a process. An executing application can create a tree of processes. For example, a parent process searches a hard drive for a specified HTML document. The HTML document name is written to a global data structure like a list, which contains all the request for documents. Once the document is located, it is removed from the request list and the path is written to another global data structure, which contains the paths of located documents. To ensure a good response to the user requests, the process has a limit of five requests pending in the list. Once the limit has been reached, two new processes are spawned to handle to work load. For each process that reaches its limits, two new processes are spawned. [Figure 3-9](#) shows a tree of processes created in this manner. A process has only one parent process, but a parent process can have many children.

Figure 3-9. A tree of processes. A process spawns two new processes if a certain condition is met.



A child process can be created with its own executable image or as a duplication of the parent process. As a duplicate of the parent, the child inherits many of the attributes of the parent, including its environment, priority and scheduling policy, resource limits, open files, and shared memory segments. If the child process advances a file's position pointer, or closes the file, this will also be seen by the parent process. If the parent allocates any additional resources after the child has been created, they are not accessible to the child. In turn, if the child process allocates any resources, they are not accessible by the parent.

Some attributes of the parent are not inherited by the child. As mentioned earlier, the child does not inherit the parent's PID or PCB. Of course, each process will have different parents. The child does not inherit any file locks created by the parent or any pending signals. Timing information such as processor usage and creation time are reset for the child process. Although these processes have this relationship, they function as separate processes. The program and stack counters operate separately. Because the data segments are copied, not shared, the child can change the values of its variables without affecting the parent's copy. The child and parent share the code segment and execute the instructions immediately following the system call that creates the child process. They do not execute those instructions in lock step because they compete for the processor with all the other processes loaded in the memory.

Once created, the child process image can be replaced with another executable image. The code, data, and stack segments as well as its heap is over-written with the new process image. The new process preserves its PID and PPID. [Table 3-3](#) lists the attributes preserved by the new process after its executable image has been replaced. It also lists the system calls that return these attributes. The environment variables are also preserved unless new environment variables were specified at time of the executable was replaced. Files that were open before the executable was replaced will still be open afterward. The new process will create files with the same file permissions. The CPU time will not be reset.

**Table 3-3. Attributes Preserved by the New Process After Its Process Image Has Been Replaced with a New Process Image**

<b>Attributes preserved</b>	<b>Description</b>
Process ID	getpid()
Parent Process ID	getppid()
Process Group ID	getpgid()
Session membership	getsid()
Real User ID	getuid()
Real Group ID	getgid()
Supplementary Group IDs	getgroups()
Time left on an alarm signal	alarm()
Nice value	nice()
Time used so far	times()
Process signal mask	sigprocmask()
Pending signals	sigpending()
File size limit	ulimit()
Resource limit	getrlimit()
File mode creation mask	umask()
Current working directory	getcwd()
Root directory	



### 3.6.1.1 The pstree Utility

The pstree utility in the Linux environment displays a tree of processes. It shows the running processes in a tree structure. The root of the tree is the init process.

## Synopsis

```
pstree [-a] [-c] [-h | -Hpid] [-l] [-n] [-p] [-u] [-G] | -U  
[pid | user]  
pstree -V
```

These are some of the options that can be used with this utility:

- a Show command-line arguments
- h Highlight the current process and its ancestors
- H Like -h but highlight the specified process instead
- n Sort processes with the same ancestor by PID instead of by name
- p Show PIDs

[Figure 3-10](#) shows the output of pstree -h in the Linux environment.

**Figure 3-10 Output of pstree -h in the Linux environment.**

```
ka:~ # pstree -h  
init-+-aplix  
    |-atd  
    |-axmain  
    |-axnet  
    |-cron  
    |-gpm  
    |-inetd  
    |-9*[kdeinit]  
    |-kdeinit    +-kdeinit  
                |-kdeinit---bash---gimp---script-fu  
                '-kdeinit---bash    +-man---sh---sh---less  
                                     '-pstree  
    |-kdeinit---cat  
    |-kdm-+-X  
        '-kdm---kde---ksmserver  
    |-kflushd  
    |-khubd  
    |-klogd  
    |-knotify  
    |-kswapd  
    |-kupdate  
    |-login---bash  
    |-lpd  
    |-mdrecoveryd  
    |-5*[mingetty]
```

```
| -nscd ---nscd ---5*[nscd]  
| -sshd  
| -syslogd  
| -usbmgr  
| -xconsole
```

### 3.6.2 Using the fork() Function Call

The fork() call creates a new process that is a duplication of the calling process, the parent. The fork() returns two values if it succeeds, one to the parent and one to the child process. It will return 0 to the child process and return the PID of the child to the parent process. The parent and child processes continue to execute from the instruction immediately following the fork() call. If not successful, meaning no child process was created, -1 is returned to the parent process.

#### Synopsis

```
#include <unistd.h>  
  
pid_t fork(void);
```

The fork() will fail if the system does not have the resources to create another process. If there is a limit to the number of child processes the parent can spawn or the number of system-wide executing processes and that limit has been exceeded, the fork() will fail. In that case, errno will be set to indicate the error.

### 3.6.3 Using the exec Family of System Calls

The exec family of functions replaces the calling process image with a new process image. The fork() call creates a new process that is a duplication of the parent process where the exec function replaces the duplicate process image with a new one. The new process image is a regular executable file and is immediately executed. The executable can be specified as a path or a file-name. These functions can pass command-line arguments to the new process. Environment variables can also be specified. There is no return value if the function is not successful because the process image that contained the call to the exec is overwritten. If unsuccessful, -1 is returned to the calling process.

All of the exec() functions can fail under these conditions:

- Permissions are denied
  - Search permission is denied for the executable's file directory
  - Execution permission is denied for the executable file
- Files do not exist
  - Executable file does not exist
  - Directory does not exist
- File is not executable
  - File is not executable because it is open for writing by another process
  - File is not an executable file
- Problems with symbolic links

Loop exists when symbolic links are encountered while resolving the pathname to the executable

Symbolic links cause the pathname to the executable to be too long

The exec functions are used with the fork(). The fork() creates and initializes the child process with the duplicate of the parent. The child process then replaces its process image by calling an exec(). [Example 3.2](#) shows an example of the fork-exec usage.

#### Example 3.2 Using the fork-exec system calls.

```
//...
RtValue = fork();
if(RtValue == 0){
    execl("/path/direct","direct",".");
}
```

In [Example 3.2](#), the fork() function is called and the return value is stored in RtValue. If RtValue is 0, then it is the child process. The execl() function is called. The first parameter is the path to the executable module, the second parameter is the execution statement, and the third parameter is the argument. direct is utility that lists all the directories and subdirectories from a given directory. There are six versions of the exec functions, each having a different calling convention and use.

#### 3.6.3.1 execl() Functions

The execl(), execl(), execlp() functions pass the command-line arguments as a list. The number of command-line arguments should be known at compile time in order for these functions to be useful.

- `int execl(const char *path,const char *arg0,.../*,  
(char *)0 */);`

path is the pathname to the program executable. It can be specified as an absolute pathname or a relative pathname from the current directory. The next arguments are the list the command-line arguments, from arg0 to argn. There can be n number of arguments. The list is to be followed by a NULL pointer.

- `int execl(const char *path,const char *arg0,.../*,  
(char *)0 *, char *const envp[]*/*);`

This function is identical to execl() except it has an additional parameter, envp[]. This parameter contains the new environment for the new process. envp[] is a pointer to a null-terminated array of null-terminated strings. Each string has the form:

name=value

where name is the name of the environment variable and value is the string to be stored. envp[] can be assigned in this manner:

```
char *const envp[] = {"PATH=/opt/kde2:/sbin",  
"HOME=/home",NULL};
```

PATH and HOME are the environment variables in this case.

- `int execlp(const char *file,const char *arg0,.../*,  
(char *)0 */);`

file is the name of the program executable. It uses the PATH environment variable to locate the

executables. The remaining arguments list the command-line arguments as explained for `execl()` function.

These are examples of the syntax of the `execl()` functions using these arguments:

```
char *const args[] = {"direct", ".", NULL};
char *const envp[] = {"files=50", NULL};
execl("/path/direct", "direct", ".", NULL);
execle("/path/direct", "direct", ".", NULL, envp);
execlp("direct", "direct", ".", NULL);
```

Each shows the syntax of how each `execl()` function creates a process that executes the `direct` program.

## Synopsis

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execle(const char *path, const char *arg0, ... /*,
           (char *)0 *, char *const envp[] */);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const arg[]);
int execve(const char *path, char *const arg[],
           char *const envp[]);
int execvp(const char *file, char *const arg[]);
```

### 3.6.3.2 `execv()` Functions

The `execv()`, `execve()`, and `execvp()` functions pass the command-line arguments in a vector of pointers to null-terminated strings. The number of command-line arguments should be known at compile time in order for these functions to be useful. `argv[0]` is usually the execution statement.

- `int execv(const char *path, char *const arg[]);`

`path` is the pathname to the program executable. It can be specified as an absolute pathname or relative pathname to the current directory. The next argument is the null-terminated vector that contains the command-line arguments as null-terminated strings. There can be `n` number of arguments. The vector is to be followed by a `NULL` pointer. `arg[]` can be assigned in this manner:

```
char *const arg[] = {"traverse", ".", ">", "1000", NULL};
```

This is an example of a function call:

```
execv("traverse", arg);
```

In this case, the `traverse` utility will list all files in the current directory larger than 1000 bytes.

- `int execve(const char *path, char *const arg[], char *const envp[]);`

This function is identical to `execv()` except it has the additional parameter `envp[]`, described earlier.

- `int execvp(const char *file, char *const arg[]);`

file is the name of the program executable. The next argument is the null-terminated vector that contains the command-line arguments as null-terminated strings. There can be n number of arguments. The vector is to be followed by a NULL pointer.

These are examples of syntax of the `execv()` functions using these arguments:

```
char *const arg[] = {"traverse",".", ">","1000",NULL};
char *const envp[] = {"files=50",NULL};
execv("/path/traverse",arg);
execve("/path/traverse",arg,envp);
execvp("traverse",arg);
```

Each shows the syntax of how each `execv()` function creates a process that executes the traverse program.

### 3.6.3.3 Determining Restrictions on `exec()` Functions

There is a limit on the size `argv[]` and `envp[]` can be when passed to the `exec()` functions. The `sysconf()` can be used to determine the maximum size of command-line arguments plus the size of environment variables for the `exec()` functions that accept the `envp[]` parameter. To return the size, name should have the value `_SC_ARG_MAX`.

## Synopsis

```
#include <unistd.h>

long sysconf(int name);
```

Another restriction when using `exec()` and the other functions used to create processes is the maximum number of simultaneous processes allowed per user id. To return this number, name has the value `_SC_CHILD_MAX`.

### 3.6.3.4 Reading and Setting Environment Variables

Environment variables are null-terminated strings that store system-dependent information such as paths to directories that contain commands, libraries, functions, and procedures used by a process. They can also be used to transmit any useful user-defined information between the parent and the child processes. They provide a mechanism for providing specific information to a process without having it hardcoded in the program code. System environment variables are predefined and common to all shells and processes in that system. The variables are initialized by startup files. Below are the common system variables:

<code>\$HOME</code>	The absolute pathname of your home directory
<code>\$PATH</code>	A list of directories to search for commands
<code>\$MAIL</code>	The absolute pathname of your mailbox
<code>\$USER</code>	Your user id
<code>\$SHELL</code>	The absolute pathname of your login shell

`$TERM`            Your terminal type

They can be stored in a file or in an environment list. The environment list will contain pointers to null-terminated strings. The variable:

```
extern char **environ
```

points to the environment list when the process begins to execute. These strings will have the form:

```
name=value
```

as explained earlier. Processes initialized with the functions `execl()`, `execlp()`, `execv()`, and `execvp()` will inherit the environment of the parent process. Processes initialized with the functions `execve()` and `execle()` set the environment for the new process.

There are functions and utilities that can be called to examine, add, or modify environment variables. The `getenv()` is used to determine whether a specific variable has been set. The parameter `name` is the environment variable in question. The function will return `NULL` if the specified variable has not been set. If the variable has been set, the function will return a pointer to a string containing the value.

## Synopsis

```
#include <stdlib.h>

char *getenv(const char *name);
int setenv(const char *name, const char *value,
           int overwrite);
void unsetenv(const char *name);
```

For example:

```
string Path;
```

```
Path = getenv("PATH");
```

the string `Path` is assigned the value contained in the predefined environment `PATH`.

The `setenv()` is used to change or add a variable to the environment of the calling process. The parameter `name` contains the name of the environment variable to be changed or added. It is assigned the value stored in `value`. If the name variable already exists, then the value is changed to `value` if the `overwrite` parameter is nonzero. If `overwrite` is 0, the content of the specified environment variable is not modified. `setenv()` return 0 if it is successful and -1 if unsuccessful. The `unsetenv()` removes the environment variable specified by `name`.

### 3.6.4 Using `system()` to Spawn Processes

The `system()` is used to execute a command or executable program. The `system()` causes the execution of `fork-exec`, and a shell. The `system()` function executes a `fork()` and the child process calls an `exec()` with a shell that executes the given command or program.

## Synopsis

```
#include <stdlib.h>

int system(const char *string);
```

The string parameter can be a system command or the name of an executable file. If successful, the function returns the termination status of the command or return value (if any) of the program. Errors can happen at several levels, the fork() or exec() functions may fail or the shell may not be able to execute the command or program.

The function returns a value to the parent process. The function returns 127 if the exec() fails and -1 if some other error occurs. The return code of the command is returned if the function succeeds. This function does not affect the wait status of any of the children processes.

### 3.6.5 The POSIX Functions for Spawning Processes

Similar to the system() and fork-exec method of process creation, the posix\_spawn() functions create new child processes from specified process images. But the posix\_spawn() functions create child processes can be created with more fine-grained control. These functions control the attributes the child process inherits from the parent process including:

- file descriptors
- scheduling policy
- process group id
- user and group id
- signal mask

They also control whether signals ignored by the parent will be ignored by the child or reset to a default action. Controlling file descriptors allow the child process independent access to the data stream independent opened by the parent. Being able to set the child's process group id affects how the child's job control will relate to that of the parent. The child's scheduling policy can be set to be different from the scheduling policy of the parent.

## Synopsis

```
#include <spawn.h>

int posix_spawn(pid_t *restrict pid, const char *restrict path,
               const posix_spawn_file_actions_t *file_actions,
               const posix_spawnattr_t *restrict attrp,
               char *const argv[restrict],
               char *const envp[restrict]);

int posix_spawnp(pid_t *restrict pid, const char *restrict file,
                const posix_spawn_file_actions_t *file_actions,
                const posix_spawnattr_t *restrict attrp,
                char *const argv[restrict],
                char *const envp[restrict]);
```

The difference between these two functions is posix\_spawn() has a path parameter and posix\_spawnp()

has a file parameter. The path parameter in the `posix_spawn()` function is the absolute or relative pathname to the executable program file. The file parameter in the `posix_spawnp()` function is the name of the executable program. If the parameter contains a slash, then file will be used as a pathname. If not, then the path to the executable is determined by the `PATH` environment variable.

The `file_action` parameter is a pointer to a `posix_spawn_file_actions_t` structure:

```
struct posix_spawn_file_actions_t{
{
    int __allocated;
    int __used;
    struct __spawn_action *actions;
    int __pad[16];
};
```

The `posix_spawn_file_actions_t` is a data structure that contains information about the actions to be performed in the new process with respect to file descriptors. The `file_action` parameter is used to modify the parent's set of open file descriptors to a set of file descriptors for the spawned child process. This structure can contain several file action operations to be performed in the sequence in which they were added to the spawn file action object. These file action operations are performed on the open file descriptors of the parent process. These operations can duplicate, reset, add, delete or close a specified file descriptors on behalf of the child process even before it's spawned. If the `file_action` parameter is a null pointer, then the file descriptors opened by the parent process will remain open for the child process without any modifications. [Table 3-4](#) lists the functions used to add file actions to the `posix_spawn_file_actions` object.

The `attrp` parameter points to a `posix_spawnattr_t` structure:

```
struct posix_spawnattr_t
{
    short int __flags;
    pid_t __pgrp;
    sigset_t __sd;
    sigset_t __ss;
    struct sched_param __sp;
    int __policy;
    int __pad[16];
};
```

This structure contains information about the scheduling policy, process group, signals and flags for the new process. The descriptions of individual attributes are as follows:

- `__flags` Used to indicate which process attributes are to be modified in the spawned process.
- `__pgrp` The id of the process group to be joined by the new process.
- `__sd` Represents the set of signals to be forced to use default signal handling by the new process.
- `__ss` Represents the signal mask to be used by the new process.
- `__sp` Represents the scheduling parameter to be assigned to the new process.



\_\_policy Represents the scheduling policy to be used by the new process.

**Table 3-4. Functions Used to Add File Actions to the posix\_spawn\_file\_actions Object**

<b>File Action Attributes Functions</b>	<b>Descriptions</b>
<pre>int posix_spawn_file_actions_addclose (posix_spawn_file_actions_t  *file_actions, int fildes);</pre>	Adds a close() action to a spawn file action object specified by file_actions. This causes the file descriptor fildes to be closed when the new process is spawned using this file action object.
<pre>int posix_spawn_file_actions_addopen (posix_spawn_file_actions_t  *file_actions, int fildes,  const char *restrict path,  int oflag, mode_t mode);</pre>	Adds an open() action to a spawn file action object specified by file_actions. This causes the file named path with the returned file descriptor fildes to be opened when the new process is spawned using this file action object.
<pre>int posix_spawn_file_actions_adddup2 (posix_spawn_file_actions_t  *file_actions, int fildes,  int new_fildes);</pre>	Adds a dup2() action to spawn a file action object specified by file_actions. This causes the file descriptor fildes to be duplicated with the file descriptor newfildes when the new process is spawned using this file action object.
<pre>int posix_spawn_file_actions_destroy (posix_spawn_file_actions_t  *file_actions);</pre>	Destroys the specified file_actions object. This causes the object to be uninitialized. The object can then become reinitialized using posix_spawn_file_actions_init().
<pre>int posix_spawn_file_actions_destroy (posix_spawn_file_actions_t  *file_actions);</pre>	Initializes the specified file_actions object. Once initialized, it will contain no file actions to be performed.

They are bitwise-inclusive OR of 0 or more of the following:

POSIX\_SPAWN\_RESETIDS

POSIX\_SPAWN\_SETPGROUP

POSIX\_SPAWN\_SETSIGDEF

POSIX\_SPAWN\_SETSIGMASK

POSIX\_SPAWN\_SETSCHEDPARAM

POSIX\_SPAWN\_SETSCHEDULER

[Table 3-5](#) lists the functions used to set and retrieve the individual attributes contained in the posix\_spawnattr\_t structure.

**Table 3-5. Functions Used to Set and Retrieve the Individual Attributes Contained in the `posix_spawnattr_t` Structure**

<b>Spawn Process Attributes functions</b>	<b>Descriptions</b>
<code>int posix_spawnattr_getflags (const posix_spawnattr_t *restrict attr, short *restrict flags);</code>	Returns the value of the <code>__flags</code> attribute stored in the specified <code>attr</code> object.
<code>int posix_spawnattr_setflags (posix_spawnattr_t *attr, short flags);</code>	Sets the value of the <code>__flags</code> attribute stored in the specified <code>attr</code> object to <code>flags</code> .
<code>int posix_spawnattr_getpgroup (const posix_spawnattr_t *restrict attr, pid_t *restrict pgroup);</code>	Returns the value of the <code>__pgroup</code> attribute stored in the specified <code>attr</code> object and stores it in the <code>pgroup</code> parameter.
<code>int posix_spawnattr_setpgroup (posix_spawnattr_t *attr, pid_t pgroup);</code>	Sets the value of the <code>__pgroup</code> attribute stored in the specified <code>attr</code> object to the <code>pgroup</code> parameter if <code>POSIX_SPAWN_SETPGROUP</code> is set in the <code>__flags</code> attribute.
<code>int posix_spawnattr_getschedparam (const posix_spawnattr_t *restrict attr, struct sched_param *restrict schedparam);</code>	Returns the value of the <code>__sp</code> attribute stored in the specified <code>attr</code> object and stores it in the <code>schedparam</code> parameter.
<code>int posix_spawnattr_setschedparam (posix_spawnattr_t *attr const struct sched_param *restrict schedparam);</code>	Sets the value of the <code>__sp</code> attribute stored in the specified <code>attr</code> object to the <code>schedparam</code> parameter if <code>POSIX_SPAWN_SETSCHEDPARAM</code> is set in the <code>__flags</code> attribute.
<code>int posix_spawnattr_getschedpolicy (const posix_spawnattr_t *restrict attr, int *restrict schedpolicy);</code>	Returns the value of the <code>__policy</code> attribute stored in the specified <code>attr</code> object and stores it in the <code>schedpolicy</code> parameter.
<code>int posix_spawnattr_setschedpolicy (posix_spawnattr_t *attr, int schedpolicy);</code>	Sets the value of the <code>__policy</code> attribute stored in the specified <code>attr</code> object to the <code>schedpolicy</code> parameter if <code>POSIX_SPAWN_SETSCHEDULER</code> is set in the <code>__flags</code> attribute.
<code>int posix_spawnattr_getsigdefault (const posix_spawnattr_t *restrict attr, sigset_t *restrict sigdefault);</code>	Returns the value of the <code>__sd</code> attribute stored in the specified <code>attr</code> object and stores it in the <code>sigdefault</code> parameter.
<code>int posix_spawnattr_setsigdefault (posix_spawnattr_t *attr,</code>	Sets the value of the <code>__sd</code> attribute stored in the

## Spawn Process Attributes functions

## Descriptions

<pre>const sigset_t *restrict sigdefault);</pre>	specified attr object to the sigdefault parameter if POSIX_SPAWN_SETSIGDEF is set in the __flags attribute.
<pre>int posix_spawnattr_getsigmask (const posix_spawnattr_t *restrict attr, sigset_t *restrict sigmask);</pre>	Returns the value of the __ss attribute stored in the specified attr object and stores it in the sigmask parameter.
<pre>int posix_spawnattr_setsigmask (posix_spawnattr_t *restrict attr, const sigset_t *restrict sigmask);</pre>	Sets the value of the __ss attribute stored in the specified attr object to the sigmask parameter if POSIX_SPAWN_SETSIGMASK is set in the __flags attribute.
<pre>int posix_spawnattr_destroy (posix_spawnattr_t *attr);</pre>	Destroys the specified attr object. The object can then become reinitialized using posix_spawnattr_init().
<pre>int posix_spawnattr_init (posix_spawnattr_t *attr);</pre>	Initializes the specified attr object with default values for all of the attributes contained in the structure. The object can then become reinitialized using posix_spawnattr_init().

[Example 3.3](#) shows how the posix\_spawn() function can be used to create a process.

**Example 3.3 Spawning a process, using the posix\_spawn() function, that calls the ps utility.**

```
#include <spawn.h>
#include <stdio.h>
#include <errno.h>
#include <iostream>
{
    //...
    posix_spawnattr_t X;
    posix_spawn_file_actions_t Y;
    pid_t Pid;
    char *const argv[] = {"/bin/ps", "-lf", NULL};
    char *const envp[] = {"PROCESSES=2"};
    posix_spawnattr_init(&X);
    posix_spawn_file_actions_init(&Y);
    posix_spawn(&Pid, "/bin/ps", &Y, &X, argv, envp);
    perror("posix_spawn");
    cout << "spawned PID: " << Pid << endl;
    //...
    return(0);
}
```

In [Example 3.3](#), the posix\_spawnattr\_t and posix\_spawn\_file\_actions\_t objects are initialized. The posix\_spawn() function is called with the arguments: PID, the path, Y, X, and argv, which contains the

command as the first element and the argument as the second, and the envp, the environment list. If the `posix_spawn()` function is successful, then the value stored in `Pid` will be the PID of the spawned process. `perror` will display:

```
posix_spawn: Success
```

and the `Pid` is sent to output. The spawned process, in this case, executes:

```
/bin/ps -lf
```

These functions return the process id of the child process to the parent process in the `pid` parameter and returns 0 as the return value. If the function is unsuccessful, no child process is created, thus no `pid` is returned and an error value is returned as the return value of the function.

Errors can occur on three levels when using the spawn functions. An error can occur if the `file_actions` or `attr` objects are invalid. If this occurs after the function has successfully returned (the child process was spawned), then the child process may have an exit status of 127. If the spawn attribute functions cause an error, then the error produced for that particular function (listed in [Tables 3-4](#) and [3-5](#)) is returned. If the spawn function has already successfully returned, then the child process may have an exit status of 127.

Errors can also occur when attempting to spawn the child process. These errors would be the same errors produced by `fork()` or `exec()` functions. If they occur, they will be the return values for the spawn functions. If the child process produces an error, it is not returned to the parent process. In order for the parent process to be aware that the child has produced an error, other mechanisms would have to be used since it will not be stored in the child's exit status. Interprocess communication can be used or the child could set some flag visible to the parent.

### 3.6.6 Identifying the Parent and Child with Process Management Functions

There are two functions that return the calling process's PID and the parent process's PID. `getpid()` returns the process id of the calling process. `getppid()` returns the parent id of the calling process. These functions are always successful, therefore no errors are defined.

#### Synopsis

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

## 3.7 Terminating a Process

When a process is terminated, the PCB is erased and the address space and resources used by the terminated process are deallocated. An exit code is placed in its entry in the main process table. The entry is removed once the parent has accepted the exit code. The termination of the process can occur under several conditions:

- All instructions have executed.
- The process makes an explicit return or makes a system call that terminates the process.
- Child processes may automatically terminate when the parent has terminated.
- The parent sends a signal to terminate its child processes.

Abnormal termination of a process can occur when the process itself does something that it shouldn't:

- The process requires more memory than the system can provide.
- The process attempts to access resources it is not allowed to access.
- The process attempts to perform an invalid instruction or a prohibited computation.

The termination of a process can also be initiated by a user when the process is interactive.

The parent process is responsible for the termination/deallocation of its children. The parent process should wait until all its child processes have terminated. When a parent process retrieves a child process's exit code, the child process exits the system normally. The process is in a zombied state until the parent accepts the signal. If the parent never accepts the signal because it has already terminated and exited the system or because it is not waiting for the child process, the child remains in the zombied state until the init process (the original system process) accepts its exit code. Many zombied processes can negatively affect the performance of the system.

### 3.7.1 The `exit()`, `kill()` and `abort()` Calls

There are two functions a process can call for self termination, `exit()` and `abort()`. The `exit()` function causes a normal termination of the calling process. All open file descriptors associated with the process will be closed. The function will flush all open streams that contain unwritten buffered data, then the open streams are closed. The status parameter is the process's exit status. It is returned to the waiting parent process, which is then restarted. The value of status may be 0, `EXIT_FAILURE`, or `EXIT_SUCCESS`. The 0 value means the process has terminated successfully. The waiting parent process only has access to the lower 8 bits of status. If the parent process is not waiting for the process to terminate, the zombied process is adopted by the init process.

The `abort()` function causes an abnormal termination of the calling process. An abnormal termination of the process causes the same effect as `fclose()` on all open streams. A waiting parent process will receive a signal that the child process aborted. A process should only abort when it encounters an error that it cannot deal with programmatically.

#### Synopsis

```
#include <stdlib.h>

void exit(int status);
void abort(void);
```

The kill() function can be used to cause the termination of another process. The kill() function sends a signal to the processes specified or indicated by the parameter pid. The parameter sig is the signal to be sent to the specified process. The signals are listed in the header <signal.h>. To kill a process, sig has the value SIGKILL. The calling process must have the appropriate privileges to send a signal to the process, or it has a real or effective user id that matches the real or saved set user-ID of the process that receives the signal. The calling process may have permission to send only certain signals to processes and not others. If the function successfully sends the signal, 0 is returned to the calling process. If it fails, -1 is returned.

The calling process can send the signal to one or several processes under these conditions:

pid > The signal will be sent to the process whose PID is equal to the pid.  
0

pid = The signal will be sent to all the processes whose process group id is the same as the calling  
0 process.

pid = The signal will be sent to all processes for which the calling process has permission to send  
-1 that signal.

pid < The signal will be sent to all processes whose process id group is equal to the absolute value  
-1 of pid and for which the calling process has permission to send that signal.

## Synopsis

```
#include <signal.h>

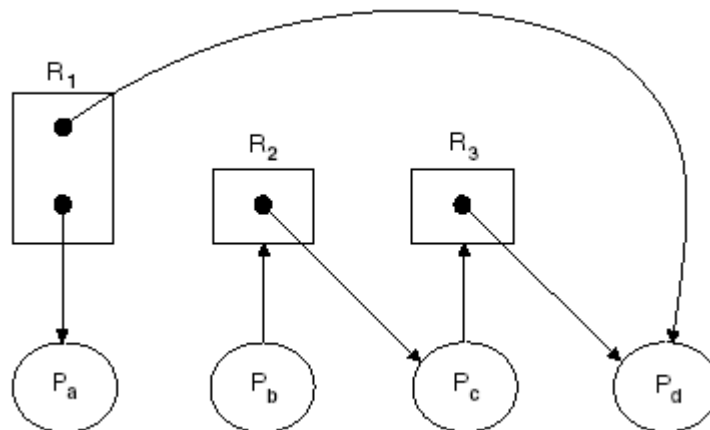
int kill(pid_t pid, int sig);
```

### 3.8 Process Resources

In order for a process to perform whatever task it is instructed to perform, it may need to write data to a file, send data to a printer, or display data to a screen. A process may need input from the user via the keyboard or input from a file. Processes can also use other processes, such as a subroutine, as a resource. Subroutines, files, semaphores, mutexes, keyboards, and display screens are all examples of resources that can be utilized by a process. A resource is anything used by a process at any given time as a source of data, a means to process, compute, or display data or information.

In order for a process to access a resource, it must first make a request to the operating system. If the resource is available, the operating system allows the process to use the resource. The process uses the resource, then releases it so it will be available to other processes. If the resource is not available, the request is denied and the process must wait. When the resource becomes available, the process is awakened. This is the basic approach to resource allocation. [Figure 3-11](#) shows a resource allocation graph, which shows which processes hold resources and which processes are requesting resources. In [Figure 3-11](#), process B makes a request for resource 2, which is held by process C. Process C makes a request for resource 3, which is held by process D.

**Figure 3-11.** A resource-allocation graph that shows which processes hold resources and which processes are requesting resources.



When more than one request to access a resource is granted, the resource is sharable, which is shown in [Figure 3-11](#) as well. Process A shares resource 1 with process D. A resource may allow many processes concurrent access or may only allow one process limited time before allowing another process access. An example of this type of shared resource is the processor. A process is assigned a processor for a short interval and then another process is assigned the processor. When only one request to access a resource is granted at a time and that occurs after the resource has been released by another process, the resource is unshared and the process has exclusive access to the resource. In a multiprocessor environment, it is important to know whether a shared resource can be accessed simultaneously or by only one process at a time in order to avoid some of the pitfalls inherent in concurrency.

Some resources can be changed or modified by a process. Other resources will not allow a process to change it. The behavior of shared modifiable or unmodifiable resources is determined by the resource type.

### S 3.1 Resource Allocation Graph

Resource allocation graphs are directed graphs that show how the resources in a system are allocated. The graph consists of a set of vertices  $V$  and a set of edges  $E$ . The set of vertices is

partitioned into two types:

$$P = \{P_1, P_2, \dots, P_n\}$$

$$R = \{R_1, R_2, \dots, R_m\}$$

Set P is the set of all the processes in the system and set R is the set of all resources in the system. A directed edge from a process to a resource is called a request edge and a directed edge from a resource to a process is called an assignment edge. These directed edges are denoted:

$P_i \rightarrow R_j$  Request edge: Process  $P_i$  requests an instance of resource type  $R_j$

$R_j \rightarrow P_i$  Assignment edge: Instance of resource type  $R_j$  has been allocated to Process  $P_i$

Each process in the resource-allocation graph is depicted as a circle and each resource is depicted as a square. Since there may be many instances of a resource type, each instance of the resource type is represented as a dot within the square. A request edge points to the perimeter of the resource square but an assignment edge originates from the dot to the perimeter of the process circle.

The resource-allocation graph in [Figure 3-11](#), depicts the following:

Sets P, R, and E

$$P = \{P_a, P_b, P_c, P_d\}$$

$$R = \{R_1, R_2, R_3\}$$

$$E = \{R_1 \rightarrow P_a, R_1 \rightarrow P_d, P_b \rightarrow R_2, R_2 \rightarrow P_c, P_c \rightarrow R_3, R_3 \rightarrow P_d\}$$

### 3.8.1 Types of Resources

There are three basic types of resources: hardware, data, and software. Hardware resources are physical devices connected to the computer. Examples of hardware resources are processors, main memory, and all other I/O devices including printers, hard disk, tape, and zip drives, monitors, keyboards, sound, network, graphic cards, and modems. All these devices can be shared by several processes.

Some hardware resources are preempted to allow different processes access. For example, a processor is preempted to allow different processes time to run. RAM is another example of a shared, preemptible resource. When a process is not executing, some of the physical page frames it occupies may be swapped out to secondary storage in order for another process to be swapped in to occupy those now-available page frames. A range of memory can only be occupied by the page frames of one process at any given time. An example of a shared, nonpreemptible resource is a printer. When a printer is shared, the jobs sent to the printer by each process is stored in a queue. Each job is printed to completion before another job starts. The printer is not preempted by any waiting printer jobs unless the current job is



canceled.

Data resources such as objects; system data such as environment variables, files, and handles, and globally defined variables such as semaphores and mutexes are all resources shared and modified by processes. Regular files and files associated with physical devices such as the printer can be opened in such a manner, restricting the type of access processes have to that file. Processes may be granted only read or write access, or read/write access. A child process inherits the parent process's resources and access rights to those resources existing at the time the child's process was created. The child process can advance the file pointer, close, modify, or overwrite the contents of a file opened by the parent. Shared memory and files with write permission require their access to be synchronized. Shared data such as semaphores or mutexes can be used to synchronize access to other shared data resources.

Shared libraries are examples of software resources. Shared libraries provide a common set of services or functions to processes. Processes can also share applications, programs, and utilities. In such a case, only one copy of the program(s) code is brought into memory. There will be separate copies of the data, one for each user (process). Program code that is not changed (also called reentrant) can be accessed by several processes simultaneously.

### 3.8.2 POSIX Functions to Set Resource Limits

POSIX defines functions that restrict a process's ability to use certain resources. The operating system sets limitations on a process's ability to utilize system resources. These resource limits affect the following:

- size of the process's stack
- size of file and core file creation
- amount of CPU usage (size of time slice)
- amount of memory usage
- number of open file descriptors

The operating system sets a hard limit on resource usage by a process. The process can set or change the soft limit of its resources but its value should not exceed the hard limit set by the operating system. A process can lower its hard limit but this value should be greater than or equal to the soft limit. When a process lowers its hard limit, it is irreversible. Only processes with special privileges can raise their hard limit.

#### Synopsis

```
#include <sys/resource.h>

int setrlimit(int resource, const struct rlimit *rlp);
int getrlimit(int resource, struct rlimit *rlp);

int getrusage(int who, struct rusage *r_usage);
```

The `setrlimit()` function is used to set limits on the consumption of specified resources. This function can set both hard and soft limits. The parameter `resource` represents the resource type. [Table 3-6](#) lists the values for `resource` with a brief description. The soft and hard limits of the specified resource are represented by the `rlp` parameter. The `rlp` parameter points to a struct `rlimit` that contains two objects of type `rlim_t`:

```

struct rlimit
{
    rlim_t rlim_cur;
    rlim_t rlim_max;
};

```

rlim\_t is an unsigned integer type. rlim\_cur contains the current or soft limit. rlim\_max contains the maximum or hard limit. rlim\_cur and rlim\_max can be assigned any value. They can also be assigned these symbolic constants defined in the header <sys/resource.h>:

RLIM_INFINITY	Indicates no limit
RLIM_SAVED_MAX	Indicates an unrepresentable saved hard limit
RLIM_SAVED_CUR	Indicates an unrepresentable saved soft limit

The soft or hard limit can be set to RLIM\_INFINITY, which means the resource is unlimited.

**Table 3-6. Values for resource**

<b>Resource definitions</b>	<b>Descriptions</b>
RLIMIT_CORE	Maximum size of a core file in bytes that may be created by a process.
RLIMIT_CPU	Maximum amount of CPU time in seconds that may be used by a process.
RLIMIT_DATA	Maximum size of a process data segment in bytes.
RLIMIT_FSIZE	Maximum size of a file in bytes that may be created by a process.
RLIMIT_NOFILE	A number 1 greater than the maximum value that the system may assign to a newly created file descriptor.
RLIMIT_STACK	Maximum size of a process stack in bytes
RLIMIT_AS	Maximum size of a process total available memory in bytes.

The getrlimit() returns the soft and hard limit of the specified resource in the rlp object. Both functions return 0 if successful and -1 if unsuccessful. [Example 3.4](#) contains an example of a process setting the soft limit for file size in bytes.

**Example 3.4 Using setrlimit() to set the soft limit for file size.**

```
#include <sys/resource.h>

//...
struct rlimit R_limit;
struct rlimit R_limit_values;

//...
R_limit.rlim_cur = 2000;
R_limit.rlim_max = RLIM_SAVED_MAX;
setrlimit(RLIMIT_FSIZE,&R_limit);
getrlimit(RLIMIT_FSIZE,&R_limit_values);
cout << "file size soft limit: " << R_limit_values.rlim_cur
    << endl;

//...
```

In [Example 3.4](#), the file size soft limit is set to 2000 bytes and the hard limit is set to the hard limit maximum. R\_limit and the RLIMIT\_FSIZE are passed to the setrlimit() function. getrlimit() are passed RLIMIT\_FSIZE and R\_limit\_value. The soft value is sent to cout.

The getrusage() function returns information about the measures of resources used by the calling process. It also returns information about the terminated child process the calling process is waiting for. The parameter who can have these values:

```
RUSAGE_SELF
RUSAGE_CHILDREN
```

If the value for who is RUSAGE\_SELF, then the information returned will pertain to the calling process. If the value for who is RUSAGE\_CHILDREN, then the information returned is pertaining to the calling process's children. If the calling process did not wait for its children, then the information pertaining to the child process is discarded. The information is returned in the r\_usage. r\_usage points to a struct rusage that contains information listed and described in [Table 3-7](#). If the function is successful, it returns 0, if unsuccessful, it returns -1.

### 3.9 What are Asynchronous and Synchronous Processes?

Asynchronous processes execute independent of each other. Process A runs until completion without any regard to process B. Asynchronous processes may or may not have a parent–child relationship. If process A creates process B, they can both execute independently but at some point the parent retrieves the exit status of the child. If they do not have a parent–child relationship, they may share the same parent.

**Table 3-7. Information Contained in struct rusage**

<b>struct rusage Attributes</b>	<b>Description</b>
struct timeval ru_utime	User time used
struct timeval ru_stime	System time used
long ru_maxrss	Maximum resident set size
long ru_maxixrss	Shared memory size
long ru_maxidrss	Unshared data size
long ru_maxisrss	Unshared stack size
long ru_minflt	Number of page claims
long ru_majflt	Number of page faults
long ru_nswap	Number of page swaps
long ru_inblock	Block input operations
long ru_oublock	Block output operations
long ru_msgsnd	Number of messages sent
long ru_msgrcv	Number of messages received
long ru_nsignals	Number of signals received
long ru_nvcsw	Number of voluntary context switches

## struct rusage Attributes

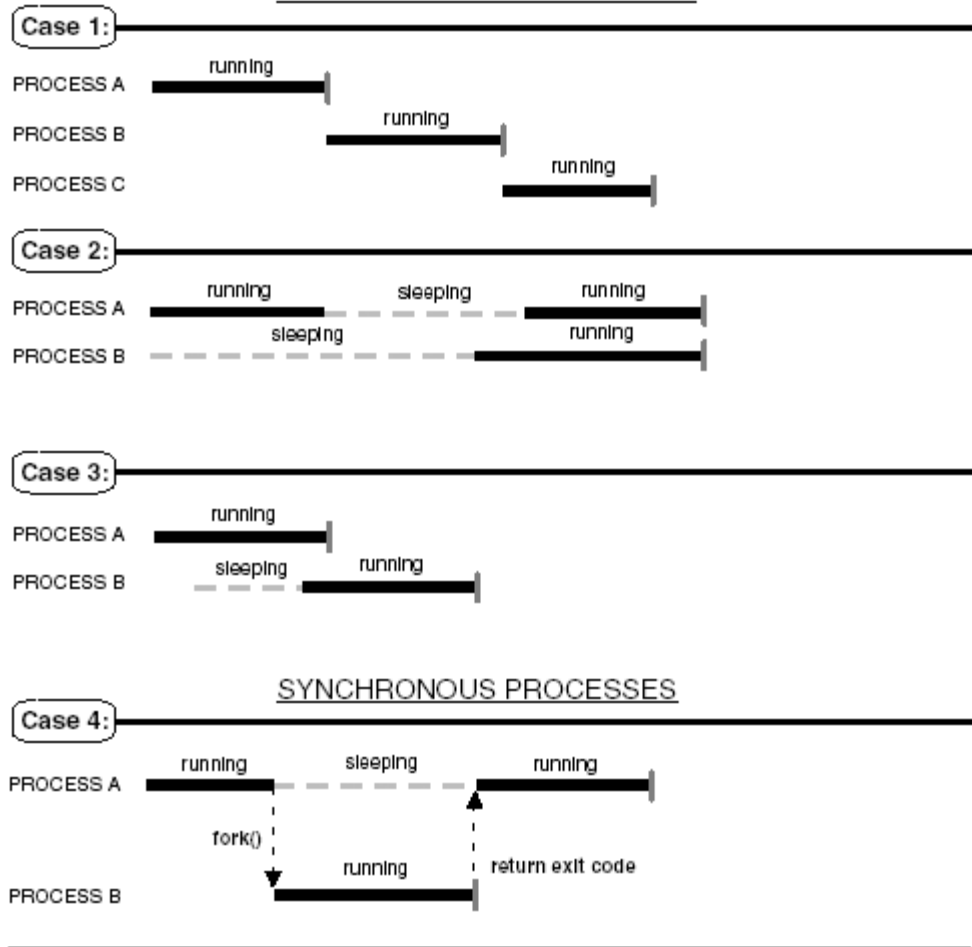
## Description

long ru\_nivcsw

Number of involuntary context switches

Asynchronous processes may execute serially, simultaneously, or overlap. These scenarios are depicted in [Figure 3-12](#). In case 1, process A runs until completion, process B runs until completion, then process C runs until completion. This is serial execution of these processes. Case 2 depicts simultaneous execution of processes. Processes A and B are active processes. While process A is running, process B is sleeping. At some point both processes are sleeping. Process B awakens before process A, process A awakens, and now both processes are running at the same time. This shows that asynchronous processes may execute simultaneously only during certain intervals of their execution. In case 3, the execution of processes A and B overlaps.

**Figure 3-12. Possible scenarios of asynchronous and synchronous processes.**



Asynchronous processes may share resources like a file or memory. This may or may not require synchronization or cooperation of the use of the resource. If the processes are executing serially (case 1), then they will not require any synchronization. For example, all three processes, A, B, and C, may share a global variable. Process A writes to the variable before it terminates, then when process B runs, it reads the data stored in the variable and before it terminates it writes to the variable. When it runs,

process C reads data from the variable. But in cases 2 and 3, the processes may attempt to modify the variable at the same time, thus requiring synchronization of its use.

For our purposes, we define synchronous processes as processes with inter-leaved execution, one process suspends its execution until another process finishes. For example, process A, the parent process, executes and creates process B, the child process. Process A suspends its execution until process B runs until completion. When process B terminates, its exit code is placed in the process table. Process A is informed that process B has terminated. Process A can resume additional processing, then terminate or it can immediately terminate. Process A and process B are synchronous processes. [Figure 3-12](#) contrasts synchronous and asynchronous execution of processes A and B.

### 3.9.1 Synchronous and Asynchronous Processes Created with `fork()`, `exec()`, `system()`, and `posix_spawn()` Functions

Processes created by `fork()`, `fork-exec`, and `posix_spawn()` functions will create asynchronous processes. When using the `fork()` function, the parent process image is duplicated. Once the child process has been created, the function returns to parent the child's PID and a return value of 0, indicating process creation was successful. The parent does not suspend execution; both processes continue to execute independently from the statement immediately preceding the `fork()`. Child processes created using the `fork-exec` combination initializes the child's process image with a new process image. The `exec()` functions do not return to the parent process unless the initialization was not successful. The `posix_spawn()` functions create the child process images and initialize it within one function call. The PID is returned to the `posix_spawn()` as well as a return value indicating if the process was spawned successfully. After `posix_spawn()` returns, both processes are executing at the same time. Processes created by the `system()` function will create synchronous processes. A shell is created that executes the system command or executable file. The parent process is suspended until the child process terminates and the `system()` call returns.

### 3.9.2 The `wait()` Function Call

Asynchronous processes can suspend execution until a child process terminates by executing the `wait()` system call. After the child process terminates, a waiting parent process collects the child's exit status, which prevents zombied processes. The `wait()` function obtains the exit status from the process table. The status parameter points to a location that contains the exit status of the child process. If the parent process has more than one child process and several of them have terminated, the `wait()` function only retrieves the exit status for one child process from the process table. If the status information is available before the execution of the `wait()` function, the function will return immediately. If the parent process does not have any children, the function returns with an error code. The `wait()` function can also be called when the calling process is to wait until a signal is delivered then perform some signal handling action.

#### Synopsis

```
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

The `waitpid()` function is the same as `wait()` except it takes an additional parameter, `pid`. The `pid` parameter specifies a set of child processes for which the exit status is retrieved. Which processes are in the set is determined by the value of `pid`:

- pid > 0     A single child process.
- pid = 0     Any child process whose group id is the same as the calling process.
- pid < -1    Any child processes whose group id is equal to the absolute value of pid.
- pid = -1    Any child processes.

The options parameter determines how the wait should behave and can have the value of the following constants defined in the header <sys/wait.h>:

- WCONTINUED    Reports the exit status of any continued child process specified by pid) whose status has not been reported since it continued.
- WUNTRACED    Reports the exit status of any child process (specified by pid) that has stopped whose status has not been reported since they stopped.
- WNOHANG        The calling process is not suspended if the exit status for the specified child process is not available.

These constants can be logically OR'ed and passed as the options parameter (e.g., WCONTINUED || WUNTRACED).

Both functions return the PID of the child process whose exit status was obtained. If the value stored in status is 0, then the child process has terminated under these conditions:

- Process returned 0 from the function main()
- Process called some version of exit() with a 0 argument
- Process was terminated because the last thread of the process terminated

[Table 3-8](#) lists the macros in which the value of the exit status can be evaluated.

**Table 3-8. Macros in Which the Value of the Exit Status Can be Evaluated**

**Macros for evaluating status     Description**

- WIFEXITED     Evaluates to nonzero if status was returned by a normally terminated child process.
- WEXITSTATUS    if WIFEXITED is nonzero, this evaluates to the low-order 8 bits of the status argument the terminated child process passed to \_exit(), exit(), or value returned from main().
- WIFSIGNALED    Evaluates to nonzero if status was returned from a child process that

**Macros for evaluating status**      **Description**

terminated because it was sent a signal that was not caught.

**WTERMSIG**      If WIFSIGNALED is nonzero, this evaluates to the number of the signal that caused the child to terminate.

**WIFSTOPPED**      Evaluates to nonzero if status was returned from a child process that currently stopped.

**WSTOPSIG**      If WIFSTOPPED is nonzero, this evaluates to the number of the signal that caused the child process to stop.

**WIFCONTINUED**      Evaluates to nonzero if status was returned from a child process that has continued from a job control stop.



### 3.10 Dividing the Program into Tasks

When considering dividing your program into multiple tasks, you are introducing concurrency into your program. In a single processor environment, concurrency is implemented with multitasking. This is accomplished by process switching. Each process executes for a short interval, then the processor is given to another process. This occurs so quickly that it gives the illusion that the processes are executing simultaneously. In a multiprocessor environment, processes belonging to a single program can all be assigned to the same processor or different processors. If the processes are assigned to different processors, then the processes will execute in parallel. Two levels for concurrent processing within an application or system are the process level and the thread level. Concurrent processing on the thread level is called multithreading, which will be discussed in the next chapter. The key to dividing your program into concurrent tasks is identifying where concurrency occurs, and where you can take advantage of it. Sometimes concurrency is not absolutely necessary. Your program may have a concurrency interpretation yet serially execute just fine. The concurrency might benefit your program with increased speed and less complexity. Some programs have natural parallelism, while others are naturally sequential by nature. Some programs have dual interpretations. When decomposing your program into functions or objects, the top-down approach is used to break down the program into functions and the bottom-up approach is used to break down the program into objects. Once this is done, it is necessary to determine which functions or objects are best served as separate programs or subprograms while others will be executed by threads. These subprograms will be executed by the operating system as processes. The separate processes perform the tasks you have designated them to do.

A program separated into tasks can execute simultaneously in three ways:

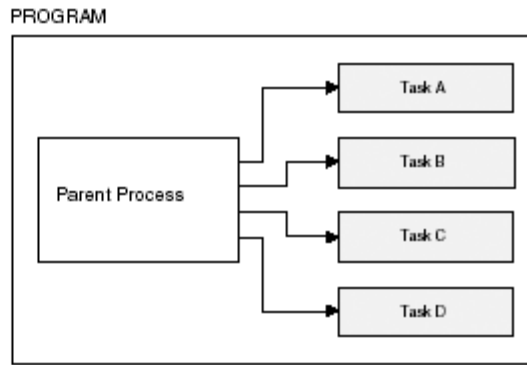
1. Divide the program into a main task that creates a number of subtasks.
2. Divide the program into a set of separate binaries.
3. Divide the program into several types of tasks in which each task type is responsible for creating only certain tasks as needed.

These approaches are depicted in [Figure 3-13](#).

Figure 3-13. Approaches that can be used to divide a program up into separate tasks.

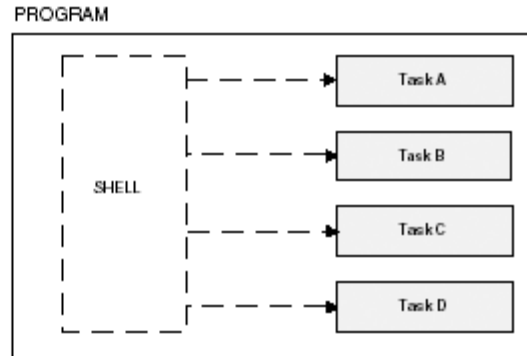
**APPROACH 1:**

Divide the program into a parent process that creates a number of child processes



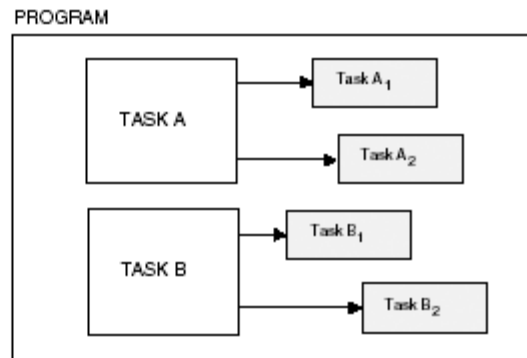
**APPROACH 2:**

Divide the program into a set of separate binaries.



**APPROACH 3:**

Divide the program into several types of processes in which each process is responsible for creating certain processes as needed.



For example, a rendering program can use these approaches. Rendering describes the process of going from a database representation of a 3D object to a shaded 2D projection on a view surface (computer screen). The image is represented as shaded polygons that exact the form of the object. The stages of the render process are shown in [Figure 3-14](#). It can be broken down into separate tasks:

**Figure 3-14. The stages of the render process.**

STAGES OF RENDERING PROCESS

vertex #	x	y	z
1	1.40000	0.00000	2.30000
2	1.40000	-0.78400	2.30000
3	0.78400	-1.40000	2.30000
4	0.00000	-1.40000	2.30000
5	1.33750	0.00000	2.53125
6	1.33750	-0.75000	2.53125
7	0.74900	-1.33700	2.53125
8	0.00000	-1.33700	2.53125
9	1.43750	0.00000	2.53125
10	1.43750	-0.90500	2.53125
.	.	.	.
.	.	.	.
.	.	.	.
306	1.42500	-0.79800	0.00000



1. Database representation of 3D object.

2. Polygon mesh model of 3D object.

3. Shaded 3D object.

1. Set up the data structure for polygon mesh models.
2. Apply linear transformations.
3. Cull back-facing polygons.
4. Perform rasterization.
5. Apply hidden surface removal algorithm.
6. Shade the individual pixels.

The first task is representing the object as an array of polygons in which each vertex in the polygon uses 3D world coordinates. The second task is applying linear transformations to the polygon mesh model. These transformations are used to position objects into a scene and to create the view point or view surface (what is seen by the observer from the view point they are observing the scene or object). The third task is culling back-facing surfaces of the objects in the scene. This means lines generated from the back portion of objects not visible from the view point are removed. This is also called back-face elimination. The fourth task is converting the vertex-based model to a set of pixel coordinates. The fifth task is removing any hidden surfaces. If there are objects interacting in the scene, objects behind others, for example, these surfaces are removed. The sixth task is shading the surfaces.

Each task is saved separately and compiled into standalone executable files. Task1, Task2, and Task3 are executed sequentially and Task4, Task5, and Task6 are executed simultaneously. In [Example 3.5](#), approach 1 is used to execute our rendering program.

**Example 3.5 Using approach 1 to create processes.**

```
#include <spawn.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>

int main(void)
{
    posix_spawnattr_t Attr;
    posix_spawn_file_actions_t FileActions;
```

```

char *const argv4[] = {"Task4",...,NULL};
char *const argv5[] = {"Task5",...,NULL};
char *const argv6[] = {"Task6",...,NULL};
pid_t Pid;
int stat;
//...

// execute first 3 tasks synchronously
system("Task1 ...");
system("Task2 ...");
system("Task3 ...");

// initialize structures
posix_spawnattr_init(&Attr);
posix_spawn_file_actions_init(&FileActions);

// execute last 3 tasks asynchronously
posix_spawn(&Pid,"Task4",&FileActions,&Attr,argv4,NULL);
posix_spawn(&Pid,"Task5",&FileActions,&Attr,argv5,NULL);
posix_spawn(&Pid,"Task6",&FileActions,&Attr,argv6,NULL);

// like a good parent, wait for all your children
wait (&stat);
wait (&stat);
wait (&stat);
return(0);
}

```

In [Example 3.5](#), from main() Task1, Task2, and Task3 are executed using the system() function. Each of these tasks is performed synchronously to the parent process. Task4, Task5, and Task6 are performed asynchronously to the parent process using posix\_spawn() functions. The ellipse (...) is used to indicate whatever files the tasks require. Parent process calls three wait() functions. Each waits for one of the Task4, Task5, and Task6 to terminate.

Using approach 2, the rendering program can be launched from a shell script. The advantage of using a shell script is all of the shell commands and operators can be used. For our render program, the & and && metacharacters are used to manage the execution of the task:

```

Task1 ... && Task2 ... && Task3
Task4 ... & Task5 ... & Task6

```

Here, Task1, Task2, and Task3 are executing sequentially under the condition the previous task executed successfully by using the && metacharacter. Task4, Task5, and Task6 executed simultaneously using the & metacharacter. The UNIX/Linux environments use metacharacters to control the way commands are executed. These are some of the metacharacters that can be used to control execution of several commands:

& Commands separated by && tokens causes the next command to be executed only if the & previous command executes successfully.

|| Commands separated by || tokens causes the next command to be executed only if the previous command fails to execute successfully.

;; Commands separated by ; tokens causes the next command to be executed next in the sequence.

& Commands separated by & tokens causes all the commands to be executed simultaneously.

Using approach 3, the tasks are categorized. When decomposing a program, it is a good technique to see if there are categories of tasks present. For example, some tasks are concerned with the user interface, creating it, extracting input from it, sending it to output, and so on. Other tasks perform computations, manage data, and so on. This is a useful technique when designing a program. It can also be used in implementing a program. In our render-program, we can group tasks into several categories:

- Tasks that perform linear transformations

Viewing transformations

Scene transformations

- Tasks that perform rasterization

Line drawing

Solid area filling

Rasterizing polygons

- Tasks that perform surface removal

Hidden surface

Back-surface elimination

- Tasks that perform shading

Pixel

Scheme

Categorizing our tasks will allow our program to be more general. Processes only create other processes of a certain category of work as needed. For example, if our program is to render a single object and not a scene, then it would not be necessary to spawn a process that performs hidden surface removal; back-surface elimination may be sufficient. If the object is not to be shaded, then it would not be necessary to spawn a task that performs shading; only line drawing rasterization would be necessary. A parent process or a shell script can be used to launch our program using approach 3. The parent can determine what type of rendering is necessary and pass that information to each of the dedicated processes so that they will know which processes to spawn. The information can also be redirected to each of the dedicated processes from the shell script. In [Example 3.6](#), approach 3 is used.

**Example 3.6 Using approach 3 to create processes. The tasks are launched from a parent process.**

```
#include <spawn.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>

int main(void)
{
    posix_spawnattr_t Attr;
    posix_spawn_file_actions_t FileActions;
    pid_t Pid;
    int stat;
```

```

//...

system("Task1 ..."); //performed regardless of the type
                      rendering used

// determine what type of rendering is needed, this can be
// obtained from the user or by performing some other type
// of analysis, communicate this to other tasks through
// arguments

char *const argv4[] = {"TaskType4",...,NULL};
char *const argv5[] = {"TaskType5",...,NULL};
char *const argv6[] = {"TaskType6",...,NULL};

system("TaskType2 ...");
system("TaskType3 ...");

// initialize structures
posix_spawnattr_init(&Attr);
posix_spawn_file_actions_init(&FileActions);

posix_spawn(&Pid,"TaskType4",&FileActions,&Attr,argv4,
            NULL);
posix_spawn(&Pid,"TaskType5",&FileActions,&Attr,argv5,
            NULL);
if(Y){
    posix_spawn(&Pid,"TaskType6",&FileActions,&Attr,
                argv6,NULL);
}
// like a good parent, wait for all your children
wait(&stat);
wait(&stat);
wait(&stat);
return(0);
}

// Each TaskType will be similar
//...

int main(int argc, char *argv[])
{
    int Rt;

    //...

    if(argv[1] == X){
        // initialize structures
        //...
        posix_spawn(&Pid,"TaskTypeX",&FileActions,&Attr,...,
                    NULL);
    }
    else{
        // initialize structures
        //...

```

```

        posix_spawn(&Pid, "TaskTypeY", &FileActions, &Attr,
                    ..., NULL);
    }

    wait(&stat);
    exit(0);
}

```

In [Example 3.6](#), each task type will determine what processes need to be spawned based on the information passed to it from the parent or shell script.

### 3.10.1 Processes Along Function and Object Lines

Processes can be spawned from functions called from `main()`, as in [Example 3.7](#).

**Example 3.7 The mainline which calls the function.**

```

int main(int argc, char *argv[])
{
    //...

    Rt = func1(X, Y, Z);

    //...
}

// This is the function definition

int func1(char *M, char *N, char *V)
{
    //...

    char *const args[] = {"TaskX", M, N, V, NULL};

    Pid = fork();
    if(Pid == 0)
    {
        exec("TaskX", args);
    }
    if(Pid > 0)
    {
        //...
    }

    wait(&stat);
}

```

In [Example 3.7](#) `func1()` is called with three arguments. These arguments are passed to the spawned process.

Processes can also be spawned from methods that belong to objects. The objects can be declared in any process, as in [Example 3.8](#).

**Example 3.8 A process declaring an object.**

```

//...

```

```

my_object MyObject;

//...

// Class declaration and definition

class my_object
{
public:

    //...
    int spawnProcess(int X);
    //...

};

int my_object::spawnProcess(int X)
{

    //...

    // posix_spawn() or system()

    //...

}

```

In [Example 3.8](#), the object can create any number of processes from whatever method necessary.

## Summary

Concurrency in a C++ program is accomplished by factoring your program into either multiple processes or multiple threads. A process is a unit of work created by the operating system. It is an artifact of the operating system where programs are artifacts of the developer. A program may consist of multiple processes that might not be associated with any particular program. Operating systems are capable of managing hundreds even thousands of concurrently loaded processes. Some information and attributes of a process are stored in the process control block (PCB) used by the operating system to identify the process. This information is needed by the operating system to manage each process. The operating system multitasks between processes by performing a context switch. It saves the current state of the executing process and its context to the PCB save area in order to restart the process the next time it is assigned to the CPU. When the process is utilizing a processor, it is in a running state. When it is waiting to use the CPU, it is in a ready state. The ps utility can be used to monitor the executing processes on the system. Processes that create other processes have a parent-child relationship with the created process. The creator of the process is the parent and the created process is the child process. Child processes inherit many attributes from the parent. The parent's key responsibility is to wait for the child process so it can exit the system. There are several system calls that can be used to create processes: fork(), fork-exec, system(), and posit\_spawn(). fork(), fork-exec(), and posix\_spawn() creates processes that are asynchronous to the parent process where system() creates a child process that is synchronous to the parent process. Asynchronous parents can call the wait() function and at that point synchronously wait for child processes to terminate or retrieve exit codes for already terminated child processes. A program can be divided into several processes. These processes can be spawned from a parent process, or launched from a shell script as separate binaries. Dedicated processes can spawn other processes as needed that only perform certain types of work. Processes can be spawned from functions or from methods.



## Chapter 4. Dividing C++ Programs into Multiple Threads

"As our computer systems become more complicated, this kind of abstraction gives us hope of being able to continue to manage them."

—Andrew Koenig and Barbara Moo, *Ruminations on C++*

In this Chapter

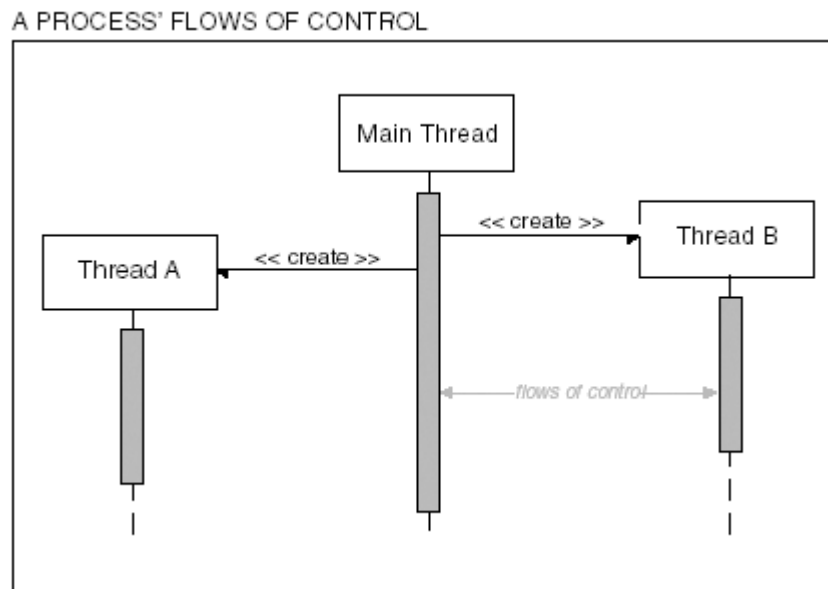
- [Threads: A Definition](#)
- [The Anatomy of a Thread](#)
- [Thread Scheduling](#)
- [Thread Resources](#)
- [Thread Models](#)
- [Introduction to the Pthread Library](#)
- [The Anatomy of a Simple Threaded Program](#)
- [Creating Threads](#)
- [Managing Threads](#)
- [Thread Safety and Libraries](#)
- [Dividing Your Program into Multiple Threads](#)
- [Summary](#)

The work of a sequential program can be divided between routines within a program. Each routine is assigned a specific task, and the tasks are executed one after another. The second task cannot start until the first task finishes, the third task cannot start until the second task finishes, and so on. This scheme works fine until performance and complexity boundaries are encountered. In some cases, the only solution to a performance problem is to allow the program to do more than one task simultaneously. In other situations the work that routines within a program have to do is so involved that it makes sense to think of the routines as mini-programs within the main program. Each mini-program executes concurrently within the main program. [Chapter 3](#) presented methods for breaking a single process up into multiple processes, where each process executes a separate program. This method allows an application to do more than one thing at a time. However, each process has its own address space and resources. Because each program is in a separate address space, communication between routines becomes an issue. Interprocess communication techniques such as pipes, fifos, and environment variables are needed to communicate between the separately executing parts. Sometimes it is desirable to have a single program do more than one task at a time without dividing the program into multiple programs. Threads can be used in these circumstances. Threads allow a single program to have concurrently executing parts, where each part has access to the same variables, constants, and address space. Threads can be thought of as mini-programs within a program. When a program is divided into separate processes, as we did in [Chapter 3](#), there is a certain amount of overhead associated with executing each of the separate programs. Threads require less overhead to execute. Threads can be thought of as lightweight processes, offering many of the advantages of processes without the communication requirements that separate processes require. Threads provide a means to divide the main flow of control into multiple, concurrently executing flows of control.

## 4.1 Threads: A Definition

A thread is a stream of executable code within a UNIX or Linux process that has the ability to be scheduled. A thread is a lighter burden on the operating system to create, maintain, and manage because very little information is associated with a thread. This lighter burden suggests that a thread has less overhead compared to a process. All processes have a main or primary thread. The main thread is a process's flow of control or thread of execution. A process can have multiple threads and therefore have as many flows of control as there are threads. Each thread will execute independently and concurrently with its own sequence of instructions. A process with multiple threads is called multithreaded. [Figure 4-1](#) shows the multiple flows of control of a process with multiple threads.

**Figure 4-1. The flows of control of a multithreaded process.**



### 4.1.1 Thread Context Requirements

All threads within the same process exist in the same address space. All of the resources belonging to the process are shared among the threads. Threads do not own any resources. Any resources owned by the process are sharable among all of the threads of that process. Threads share file descriptors and file pointers but each thread has its own program pointer, register set, state, and stack. The threads' stacks are all within the stack segment of its process. The data segment of the process is shared with its thread. A thread can read and write to the memory locations of its process and the main thread has access to the data. When the main thread writes to memory, any of the child threads can have access to the data. Threads can create other threads within the same process. All the threads in a single process are called peers. Threads can also suspend, resume, and terminate other threads within its process.

Threads are executing entities that compete independently for processor usage with threads of the same or different processes. In a multiprocessor system, threads within the same process can execute simultaneously on different processors. The threads can only execute on processors assigned to that particular process. If processors 1, 2, and 3 are assigned to process A, and process A has three threads, then a thread will be assigned to each processor. In a single processor environment, threads compete for processor usage. Concurrency is achieved through context switching. Context switches take place when the operating system is multitasking between tasks on a single processor. Multitasking allows more than one task to execute at the same time on a single processor. Each task executes for a designated time interval. When the time interval has expired or some event occurs, the task is removed from the

processor and another task is assigned to it. When threads are executing concurrently within a single process, then the process is multithreaded. Each thread executes a subtask allowing these subtasks of the process to execute independently without regard to the process's main flow of control. With multithreading, the threads can compete for a single processor or be assigned to different processors. In any case, a context switch occurring between threads of the same process requires fewer resources than a context switch occurring between threads of different processes. A process uses many system resources to keep track of its information and a process context switch takes time to manage that information. Most of the information contained in the process context describes the address space of the process and resources owned by the process. When a switch occurs between threads in different address spaces, a process context switch must take place. Since threads within the same process do not have their own address space or resources, less information tracking is needed. The context of a thread only consists of an id, a stack, a register set, and a priority. The register set contains the program or instruction pointer and the stack pointer. The text of a thread is contained in the text segment of its process. A thread context switch will take less time and use fewer system resources.

#### **4.1.2 Threads and Processes: A Comparison**

There are many aspects of a thread that are similar to a process. Threads and processes have an id, a set of registers, a state, a priority, and adhere to a scheduling policy. Like a process, threads have attributes that describe it to the operating system. This information is contained in a thread information block similar to a process information block. Threads and child processes share the resources of its parent process. The resources opened by the process (main thread) are immediately accessible to the threads and child processes of the parent process. No additional initialization or preparation is needed. Threads and child processes are independent entities from its parent or creator and compete for processor usage. The creator of the process or thread exercises some control over the child process or thread. The creator can cancel, suspend, resume, or change the priority of the child process or thread. A thread or process can alter its attributes and create new resources but cannot access the resources belonging to other processes. However, threads and processes differ in several ways.

##### **4.1.2.1 Differences between Threads and Processes**

The major difference between threads and processes is each process has its own address space and threads don't. If a process creates multiple threads, all the threads will be contained in its address space. This is why they share resources so easily and interthread communication is so simple. Child processes have their own address space and a copy of the data segment. Therefore, when a child changes its variables or data, it does not affect the data of its parent process. A shared memory area has to be created in order for parent and child processes to share data. Interprocess communication mechanisms, such as pipes and fifos, are used to communicate or pass data between them. Threads of the same process can pass data and communicate by reading and writing directly to any data that is accessible to the parent process.

##### **4.1.2.2 Threads Controlling Other Threads**

Whereas processes can only exercise control over other processes in which it has a parent-child relationship, threads within a process are considered peers and are on an equal level regardless of who created whom. Any thread that has access to the thread id of another thread in the same process can cancel, suspend, resume, or change the priority of that thread. In fact, any thread within a process can kill the process by canceling the main or primary thread. Canceling the main thread would result in terminating all the threads of the process—killing the process. Any changes to the main thread may affect all the threads of the process. When changing the priority of the process, all the threads within the process that inherited that priority and have not changed its priorities would also be altered. [Table 4-1](#)

summarizes the similarities and differences between threads and processes.

### 4.1.3 Advantages of Threads

There are several advantages of using multiple threads to manage the subtasks of an application as compared to using multiple processes. These advantages include:

- Less system resources needed for context switching
- Increased throughput of an application
- No special mechanism required for communication between tasks
- Simplify program structure

**Table 4-1. Similarities and Differences between Threads and Processes**

#### **Similarities Between Threads and Processes      Differences Between Threads and Processes**

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>• Both have an id, set of registers, state, priority, and scheduling policy.</li><li>• Both have attributes that describe the entity to the OS.</li><li>• Both have an information block.</li><li>• Both share resources with the parent process.</li><li>• Both function as independent entities from the parent process.</li><li>• The creator can exercise some control over the thread or process.</li><li>• Both can change their attributes.</li><li>• Both can create new resources.</li><li>• Neither can access the resources of another process.</li></ul> | <ul style="list-style-type: none"><li>• Threads share the address space of the process that created it; processes have their own address.</li><li>• Threads have direct access to the data segment of its process; processes have their own copy of the data segment of the parent process.</li><li>• Threads can directly communicate with other threads of its process; processes must use interprocess communication to communicate with sibling processes.</li><li>• Threads have almost no overhead; processes have considerable overhead.</li><li>• New threads are easily created; new processes require duplication of the parent process.</li><li>• Threads can exercise considerable control over threads of the same process; processes can only exercise control over child processes.</li><li>• Changes to the main thread (cancellation, priority change, etc.) may affect the behavior of the other threads of the process; changes to the parent process does not affect child processes.</li></ul> |
|--|---|

#### 4.1.3.1 Context Switches during Low Processor Availability

When creating a process, the main thread may be the only thread needed to carry out the function of the process. If the process has many concurrent subtasks, multiple threads can provide asynchronous execution of the subtasks with less overhead for context switching. When processor availability is low or there is only a single processor, concurrently executing processes involve heavy overhead because of the context switching required. Under the same condition using threads, a process context switch would only occur when a thread from a different process is the next thread to be assigned the processor. Less overhead means less system resources used and less time taken for context switching. Of course, if there are enough processors to go around then context switching is not an issue.

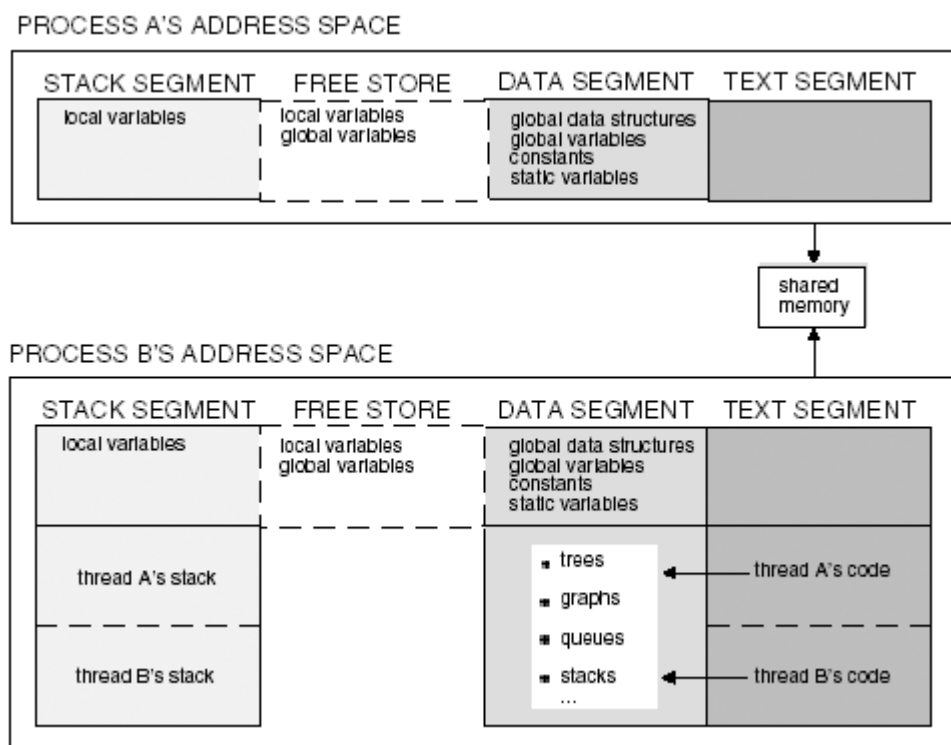
#### 4.1.3.2 Better Throughput

Multiple threads can increase the throughput of an application. With one thread, an I/O request would halt the entire process. With multiple threads, as one thread waits for an I/O request, the application can continue executing. As one thread is blocked, another can execute. The entire application does not wait for each I/O request to be filled. Other tasks can be performed that does not depend on the blocked thread.

#### 4.1.3.3 Simpler Communication between Concurrently Executing Parts

Threads do not require special mechanisms for communication between subtasks. Threads can directly pass and receive data from other threads. This also saves system resources that would have to be used in the setup and maintenance of special communication mechanisms if multiple processes were used. Threads communicate by using the memory shared within the address space of the process. Processes can also communicate by shared memory but processes have separate address spaces and therefore the shared memory exists outside the address space of both processes. This increases the time and space used to maintain and access the shared memory. [Figure 4-2](#) illustrates the communication between processes and threads.

Figure 4-2. Communication between threads of a single process and communication between multiple processes.



#### 4.1.3.4 Simplify Program Structure

Threads can be used to simplify the program structure of an application. Each thread is assigned a subtask or subroutine for which it is responsible. The thread will independently manage the execution of the subtask. Each thread can be assigned a priority reflecting the importance of the subtask it is executing to the application. This will result in more easily maintained code.

#### 4.1.4 Disadvantages of Threads

The easy accessibility threads have to the memory of a process has its disadvantages. For example:

- Threads can easily pollute address space of a process.
- Threads will require synchronization for concurrent read/write access to memory.
- One thread can kill the entire process or program.
- Threads only exist within a single process and are therefore not reusable.

##### 4.1.4.1 Threads Can Corrupt Process Data Easier

It's easier for threads to corrupt the data of a process through data race because multiple threads have write access to the same piece of data. This is not so with processes. Each process has its own data and other processes don't have access unless special programming is done. The separate address spaces of processes protect the data. The fact that threads share the same address space exposes the data to corruption. For example, a process has three threads, A, B, and C. Threads A and B write to a memory location and Thread C reads the value and uses it in a calculation. Threads A and B may both attempt to write to the memory location at the same time. Thread B overwrites the data written by Thread A before Thread C gets a chance to read it. The threads need to be synchronized so that Thread C can read the data deposited in the memory location by Thread A before Thread B overwrites it. Synchronization is needed to prevent either thread from overwriting the values before the data is used. The issues of synchronization between threads will be discussed in [Chapter 5](#).

##### 4.1.4.2 One Bad Thread Can Kill the Entire Program

Since threads don't have their own address space, they are not isolated. If a thread causes a fatal access violation, this may result in the termination of the entire process. Processes are isolated. If one process corrupts its address space, the problems are restricted to that process. A process can have an access violation that causes the process to terminate and all of the other processes will continue executing if the violation isn't too bad. Data errors can be restricted to a single process. Errors caused by a thread are more costly than errors caused by processes. Threads can create data errors that affect the entire memory space of all the threads. Processes can protect its resources from indiscriminate access by other processes. Threads share resources with all the other threads in the process. A thread that damages a resource affects the whole process or program.

##### 4.1.4.3 Threads are Not as Reusable by Other Programs

Threads are dependent and cannot be separated from the process in which they reside. Processes are more independent than threads. An application can divide tasks among many processes and those processes can be packaged as modules that can be used in other applications. Threads cannot exist outside the process that created it and therefore are not reusable. [Table 4-2](#) lists the advantages and disadvantages of threads.

**Table 4-2. Advantages and Disadvantages of Threads**

**Advantages of Threads**

- Less system resources needed for context switching
- Increased throughput of an application
- No special mechanism required for communication between tasks
- Simplification of program structure

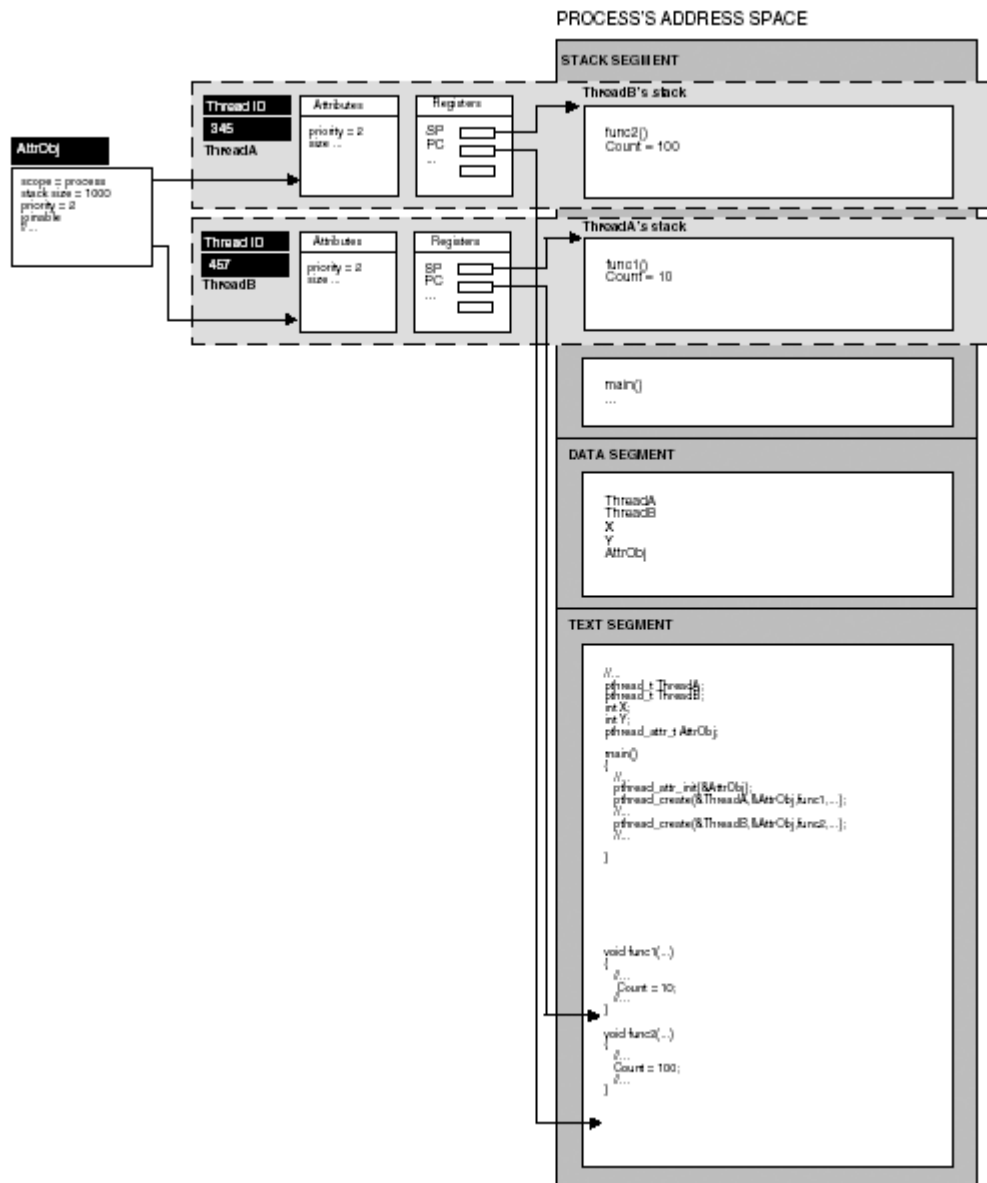
**Disadvantages of Threads**

- Require synchronization for concurrent read/write access to memory
- Can easily pollute address space of its process
- Only exist within a single process and therefore not reusable

## 4.2 The Anatomy of a Thread

The layout of a thread is embedded in the layout of a process. As discussed in [Chapter 3](#), a process has a code, data, and stack segment. The thread shares the code and data segment with the other threads of the process. Each thread has its own stack allocated in the stack segment of the process's address space. The thread's stack size is set when the thread is created. If the creator of the thread does not specify the size of the thread's stack, a default size is assigned by the system. The default size of the stack is system dependent and will depend on the maximum number of threads a process can have, the allotted size of a process's address space, and the space used by system resources. The thread's stack size must be large enough for any functions called by the thread, any code that is external to the process like library code, and local variable storage. A process with multiple threads should have a stack segment large enough for all of its thread's stacks. The address space allocated to the process limits the stack size, thus limiting the size possible for each thread's stack. [Figure 4-3](#) shows the layout of a process that contains multiple threads.

Figure 4-3. Layout of a process that contains multiple threads.





In [Figure 4-3](#), the process contains two threads and the thread's stacks are located in the stack segment of the process. Each thread executes different functions: thread A executes function 1 and thread B executes function 2.

### 4.2.1 Thread Attributes

The attributes of a process are what describe the process to the operating system. The operating system uses this information to manage processes and distinguish one process from another. The process shares almost everything with its thread including its resources and environment variables. The data segment, text segment, and all resources are associated with the process and not the threads. Everything a thread needs to operate is supplied and defined by the process. What distinguishes threads from one another is the id, the set of registers that defines the state of the thread, its priority, and its stack. These attributes are what give each thread their identity. Like the process, the information about a thread is stored in data structures and returned by functions supplied by the operating system. Some information about a thread is contained in a structure called the thread information block, created at the time the thread is created.

The thread id is a unique value that identifies each thread during its lifetime in a process. The priority of a thread determines which threads are given preferential access to an available processor at a given time. The state of the thread is the condition a thread is in at any given time. The set of registers for a thread includes the program counter and the stack pointer. The program counter contains the address of the instruction the thread is to execute and the stack pointer points to the top of the thread's stack.

The POSIX thread library defines a thread attribute object that encapsulates the properties of the thread accessible and modifiable to the creator of the thread. The thread attribute defines the following attributes:

- scope
- stack size
- stack address
- priority
- detached state
- scheduling policy and parameters

A thread attribute object can be associated with one thread or multiple threads. When an attribute object is used, it is a profile that defines the behavior of a thread or group of threads. All the threads that use the attribute object take on all the properties defined by the attribute object. [Figure 4-3](#) also shows the attributes associated with each thread. As you can see, both threads A and B share an attribute object but they maintain their separate thread ids and set of registers. Once the attribute object is created and initialized, it can be referenced in any calls to the thread creation function. Therefore, a group of threads can be created that has a "small stack, low priority" or "large stack, high priority and detached." Detached threads are threads that are not synchronized with other threads in the process. In other words, there are no threads waiting for the detached thread to exit. Therefore, once the thread exits, its resources, namely thread id, can be instantly reused. Several methods can be invoked to set and retrieve the values of these attributes. Once a thread is created, its attributes cannot be changed while the thread is in use.

The scope attribute describes which threads a particular thread will compete with for resources. Threads contend for resources within two contention scopes: process scope (threads of the same process) and system scope (all threads in the system). Threads compete with threads within the same process for file descriptors while threads with system-wide contention scope compete for resources that are allocated

across the system (e.g., real memory). Threads compete with threads that have process scope and threads from other processes for processor usage depending on the contention scope and the allocation domains (the set of processors to which it is assigned). A thread that has system scope will be prioritized and scheduled with respect to all of the system-wide threads. [Table 4-3](#) lists the settable properties for the POSIX thread attribute object with a brief description.

### 4.3 Thread Scheduling

When a process is scheduled to be executed, it is the thread that utilizes the processor. If the process has only one thread, it is the primary thread assigned to a processor. If a process has multiple threads and there are multiple processors, all of the threads are assigned to a processor. Threads compete for processor usage either with all the threads from active processes in the system or just the threads from a single process. The threads are placed in the ready queues sorted by their priority value. The threads in the queue with the same priority are scheduled to processors according to a scheduling policy. When there are not enough processors to go around, then a thread with a higher priority can preempt an executing thread. If the newly active thread is of the same process as the preempted thread, then the context switch is between threads. If the newly active thread is of another process, a process context switch occurs and then the thread context switch is performed.

**Table 4-3. Settable Properties for the Thread Attribute Object**

<b>Settable Thread Functions Attributes</b>	<b>Description</b>
detachstate int pthread_attr_setdetachstate(pthread_attr_t *attr,int detachstate);	The detachstate attribute controls whether the newly created thread is detachable. If detached, the thread's flow of control cannot be joined to any thread.
guardsize int pthread_attr_setguardsize(pthread_attr_t *attr,size_t guardsize);	The guardsize attribute controls the size of the guard area for the newly created thread's stack. It creates a buffer zone the size of guardsize at the overflow end of the stack.
inheritsched int pthread_attr_setinheritsched(pthread_attr_t *attr,int inheritsched);	The inheritsched attribute determines how the scheduling attributes of the newly created thread will be set. It determines whether the new thread's scheduling attributes are inherited from the creating thread or set by an attribute object.
param int pthread_attr_setschedparam(pthread_attr_t *restrict attr,const struct sched_param *restrict param);	The param attribute is a structure that can be used to set the priority of the newly created thread.

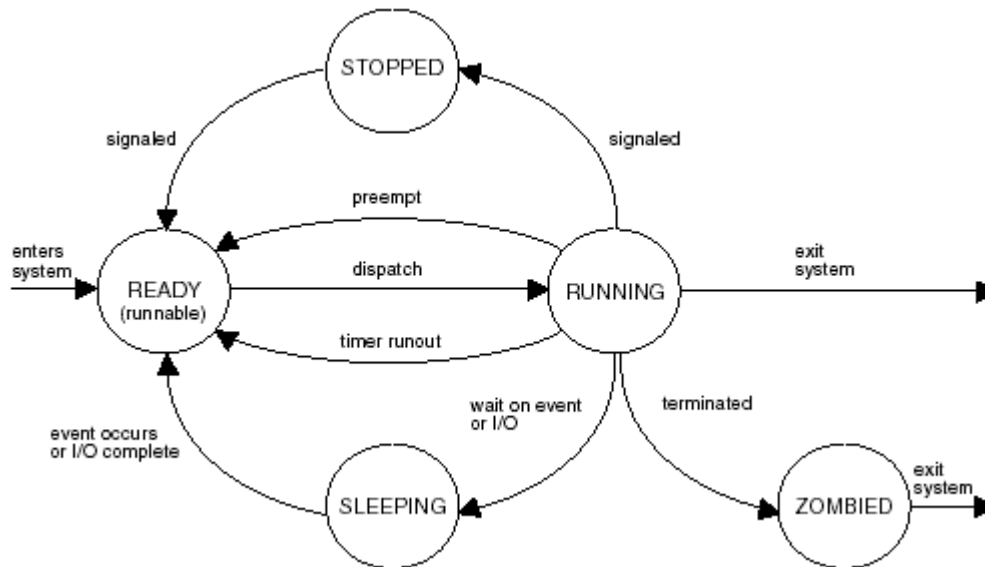
Settable Thread Functions Attributes		Description
schedpolicy	int pthread_attr_setschedpolicy(pthread_attr_t *attr,int policy);	The schedpolicy determines the scheduling policy of the newly created thread.
contentionscope	int pthread_attr_setscope(pthread_attr_t *attr,int contentionscope);	The contentionscope attribute determines which set of threads the newly created thread will compete with for processor usage. A process scope means the thread will compete with the set of threads of the same process; system scope means the thread will compete with system-wide threads (this includes threads from other processes).
stackaddr stacksize	int pthread_attr_setstack(pthread_attr_t *attr,void *stackaddr,size_t stacksize);	The stackaddr and stacksize attributes determine the base address and minimum size in bytes of the stack for the newly created thread.
stackaddr	int pthread_attr_setstackaddr(pthread_attr_t *attr,void *stackaddr);	The stackaddr attribute determines the base address of the stack for the newly created thread.
stacksize	int pthread_attr_setstacksize(pthread_attr_t *attr,size_t stacksize);	The stacksize attribute determines the minimum size in bytes of the stack for the newly created thread.

### 4.3.1 Thread States

Threads have the same states and transitions mentioned in [Chapter 3](#) that processes have. [Figure 4-4](#) is a duplication of the state diagram [3.4](#) from [Chapter 3](#). To review, there are four commonly implemented states: runnable, running (active), stopped, and sleeping (blocked). A thread state is the mode or condition a thread is in at any given time. A thread is in a runnable state when it is ready for execution. All runnable threads are placed in a ready queue with other threads with the same priority that are ready to be executed. When a thread is selected and assigned to a processor, the thread is in the running state. A thread is preempted once it has executed for its time slice or when a thread of higher priority becomes runnable. The thread is then placed back into the ready queue. A thread is in the sleeping state if it is waiting for an event to occur or I/O request to complete. A thread is stopped when it receives a signal to stop executing. It remains in that state until it receives a signal to continue. Once the signal is received, the thread moves from the stopped to a runnable state. As the thread moves from one state to another, it undergoes a state transition that signals some event has occurred. When a thread changes from the runnable to the running state it is because the system has selected that thread to run—the

thread has been dispatched. A thread is preempted if its makes an I/O request or some other request of the kernel or for some external reason.

**Figure 4-4. Thread states and transitions.**

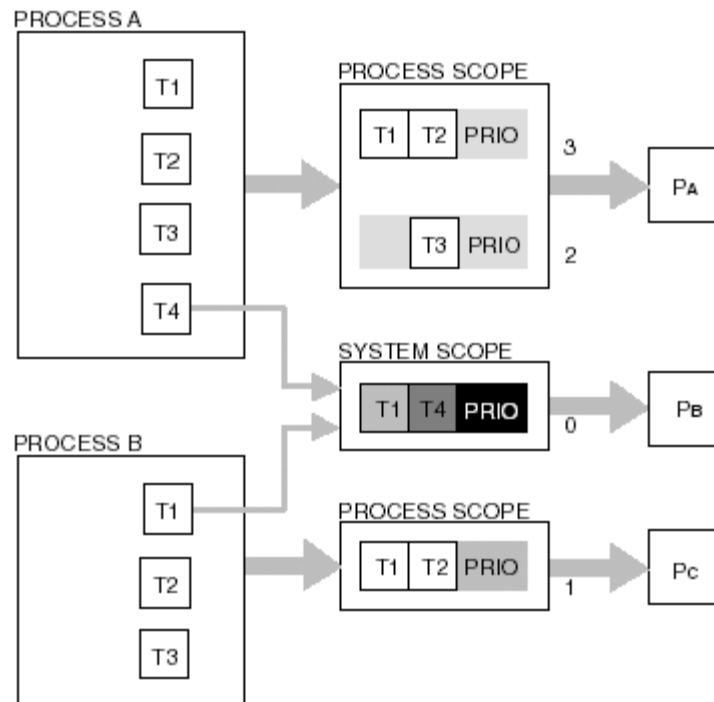


One thread can determine the state of an entire process. The state of a process with one thread is synonymous with the state of its primary thread. If the primary thread is sleeping, the process is sleeping. If the primary thread is running, the process is running. For a process that has multiple threads, all threads of the process would have to be in a sleeping or stopped state in order to consider the whole process sleeping or stopped. On the other hand, if one thread is active (runnable or running) then the process is considered active.

### 4.3.2 Scheduling and Thread Contention Scope

The contention scope of the thread determines which set of threads a thread will compete with for processor usage. If a thread has process scope, it will only compete with the threads of the same process for processor usage. If the thread has system scope, it will compete with its peers and with threads of other processes for processor usage. For example, in [Figure 4-5](#), there are two processes in a multiprocessor environment of three processors. Process A has four threads and Process B has three threads. Process A has three threads that have process scope and one thread with system scope. Process B has two threads with process scope and one thread with system scope. Process A's threads with process scope competes for processor A and Process B's threads with process scope compete for processor C. Process A and B's threads with system scope compete for processor B.

**Figure 4-5. Scheduling with process and system scope threads in a multiprocessor environment.**



NOTE:

Threads should have system scope when modeling true real-time behavior in your application.

### 4.3.3 Scheduling Policy and Priority

The scheduling policy and priority of the process belong to the primary thread. Each thread can have its own scheduling policy and priority separate from the primary thread. Threads have an integer priority value that has a maximum and minimum value. A priority scheme is used to determine which thread is assigned the processor. Each thread has a priority and the thread with the highest priority is executed before the threads of lower priority. When threads are prioritized, tasks that require immediate execution or response from the system are allotted the processor time it requires. In a preemptive operating system, executing threads are preempted if a thread of higher priority and the same contention scope is available. For example, in [Figure 4-5](#), threads with process scope compete for the processor with threads of the same process that also have process scope. Process A has two threads with priority 3 in which one is assigned the processor. Once the thread with priority 2 becomes runnable, the active thread is preempted and the processor is given to the thread with higher priority. Yet, in Process B, it has two process scope threads that have priority 1, a higher priority than 2. One thread is assigned the processor. Although the other thread with priority 1 is runnable, it does not preempt Process A's thread with priority 2. The thread with system scope and a lower priority is not preempted by any of the threads of Process A or B. They only compete for processor usage with other threads that have system scope.

As discussed in [Chapter 3](#), the ready queues are organized as a sorted list in which each element is a priority level. Each priority level in the list is a queue of threads with the same priority level. All threads of the same priority level are assigned to the processor using a scheduling policy: FIFO, round-robin, or other. With the FIFO (First-In, First-Out) scheduling policy, when the time slice expires the thread is placed at the head of the queue of its priority level. Therefore, the thread will run until it completes execution, it sleeps, or it receives a signal to stop. When a sleeping thread is awakened, it is

placed at the end of the queue of its priority level. Round-robin scheduling is similar to FIFO scheduling except the thread is placed at the end of the queue when the time slice expires and the processor is given to the next thread in the queue.

The round-robin scheduling policy considers all threads to be of equal priority and each thread is given the processor for only a time slice. Task executions are interweaved. For example, a program that searches files for specified keywords is divided into two threads. One thread, thread 1, searches for all files with a specified file extension and places the path of the file into a container. Another thread, thread 2, extracts the name of the files from the container, searches each file for the specified keywords, then writes the name of the files that contains all the keywords to a file. If the threads used a round-robin scheduling policy with a single processor, thread 1 would use its time slice to find files and insert the paths into the container. Thread 2 would use its time slice to extract file names and then perform the keyword search. In a perfect world, this interweaves the execution of threads 1 and 2. But thread 2 may execute first when there are no files in the container or thread 1 may only get as far as finding a file, the time slice expiring before it had a chance to insert the file name into the container. This situation requires synchronization, which will be discussed briefly later in this chapter and in [Chapter 5](#). The FIFO scheduling policy allows each thread to run until execution is complete. Using the same example, thread 1 would have time to locate all the files and insert the paths into the container. Thread 2 would then extract the filenames and perform its keyword search on each file. In a perfect world, this would be the end of the program. But thread 2 may be assigned to a processor before thread 1 and there would be no files in the container to search. Thread 1 would then execute, locate, and insert file names into the container but no keyword search would be performed. The program would fail. With FIFO scheduling, there is no interweaving of the execution of these tasks. A thread assigned to a processor dominates the processor until it completes execution. This scheduling policy can be used for applications where a set of threads need to complete as soon as possible. The "other" scheduling policy can be user-defined customization of a scheduling policy. For example, the FIFO scheduling policy can be customized to allow random unblocking of threads.

#### **4.3.3.1 Changing Thread Priority**

The priorities of threads should be changed in order to speed up the execution of threads on which other threads depend. They should not be changed in order for a specific thread to get more processor time. This will affect the overall performance of the system. High-priority class threads receive more processor time than threads of a lower class because they are executed more frequently. Threads of higher priority will dominate the processor, preventing other threads of lower priority valuable processor time. This is called starvation. Systems that use dynamic priority mechanisms respond to this situation by assigning priorities that last for short periods of time. The system adjusts the priority of threads in order for threads of lower priority execution time. This will improve the overall performance of the system.

The temptation to ensure that a process or specific thread runs to completion is to give it the highest priority but this will affect the overall performance of the system. Such threads may preempt communications over networks, causing the loss of data. Threads that control the user interface may be drastically affected, causing the keyboard, mouse, and screen response to be sluggish. Some systems prevent user processes and threads from having a higher priority than system processes. Otherwise, system processes or threads would be prevented from responding to critical system changes. Generally speaking, most user processes and threads fall in the category of normal or regular priority.

## 4.4 Thread Resources

Threads share most of its resources with other threads of the same process. Threads do own resources that define the thread's context. This includes the thread id, set of registers including the stack pointer and program counter, and stack. Threads must share other resources such as the processor, memory, and file descriptors required in order for it to perform its task. File descriptors are allocated to each process separately and threads of the same process compete for access to these descriptors. In memory, the processor, and other globally allocated resources, threads contend with other threads of its process as well as the threads of other processes for access to these resources.

A thread can allocate additional resources such as files or mutexes, but they are accessible to all the threads of the process. There are limits on the resources that can be consumed by a single process. Therefore, all the threads in combination must not exceed the resource limit of the process. If a thread attempts to consume more resources than the soft resource limit defines, it is sent a signal that the process's resource limit has been reached. Threads that allocate resources must be careful not to leave resources in an unstable state when they are canceled. A thread that has opened a file or created a mutex may be terminated, leaving the file open or the mutex locked. If the file has not been properly closed and the application is terminated, this may result in damage to the file or loss of data. A thread terminating after locking a mutex prevents access to whatever critical section that mutex is protecting. Before it terminates, a thread should perform some cleanup, preventing these unwanted situations from occurring.

## 4.5 Thread Models

The purpose of a thread is to perform work on behalf of the process. If a process has multiple threads, each thread performs some subtask as part of the overall task to be performed by the process. Threads are delegated work according to a specific strategy or approach that structures how delegation is implemented. If the application models some procedure or entity, then the approach selected should reflect that model. Some common models are:

- delegation (boss–worker)
- peer-to-peer
- pipeline
- producer-consumer

Each model has its own WBS (Work Breakdown Structure) that determines who is responsible for thread creation and under what conditions threads are created. For example, there is a centralized approach where a single thread creates other threads and delegates work to each thread. There is an assembly-line approach where threads perform different work at different stages. Once the threads are created, they can perform the same task on different data sets, different tasks on the same data set, or different tasks on different data sets. Threads can be categorized to only perform certain types of tasks. For example, there can be a group of threads that only perform computations, process input, or produce output.

It may be true that what is to be modeled is not homogeneous throughout the process and it may be necessary to mix models. In [Chapter 3](#), we discussed a rendering process. Tasks 1, 2, and 3 were performed sequentially and tasks 4, 5, and 6 can be performed simultaneously. Each task can be executed by a different thread. If multiple images were to be rendered, threads 1, 2, and 3 can form the pipeline of the process. As thread 1 finishes, the image is passed to thread 2 while thread 1 performs its work on the next image. As these images are buffered, threads 4, 5, and 6 can use a workpile approach. The thread model is a part of the structuring of parallelism in your application where each thread can be

executing on a different processor. [Table 4-4](#) lists the thread models with a brief description.

**Table 4-4. Thread Models**

<b>Thread Models</b>	<b>Description</b>
Delegation model	A central thread (boss) creates the threads (workers), assigning each worker a task. Each worker is assigned a task by the boss thread. The boss thread may wait until each thread completes that task.
Peer-to-peer model	All the threads have an equal working status. Threads are called peer threads. A peer thread creates all the threads needed to perform the tasks but performs no delegation responsibilities. The peer threads can process requests from a single input stream shared by all the threads or each thread may have its own input stream.
Pipeline	An assembly-line approach to processing a stream of input in stages. Each stage is a thread that performs work on a unit of input. When the unit of input has been through all the stages, then the processing of the input has been completed.
Producer–consumer model	A producer thread produces data to be consumed by the consumer thread. The data is stored in a block of memory shared by the producer and consumer threads.

### **4.5.1 Delegation Model**

In the delegation model, a single thread (boss) creates the threads (workers) and assigns each a task. It may be necessary for the boss thread to wait until each worker thread completes its task. The boss thread delegates the task each worker thread is to perform by specifying a function. As each worker is assigned its task, it is the responsibility of each worker thread to perform that task and produce output or synchronize with the boss or other thread to produce output.

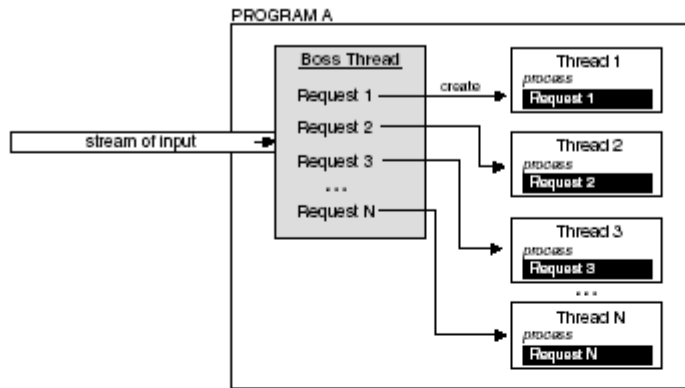
The boss thread can create threads as a result of requests made to the system. The processing of each type of request can be delegated to a thread worker. In this case, the boss thread executes an event loop. As events occur, thread workers are created and assigned their duties. A new thread is created for every new request that enters the system. Using this approach may cause the process to exceed its resource or thread limits. Alternatively, a boss thread can create a pool of threads that are reassigned new requests. The boss thread creates a number of threads during initialization and then each thread is suspended until a request is added to their queue. As requests are placed in the queue, the boss thread signals a worker thread to process the request. When the thread completes, it dequeues the next request. If none are available, the thread suspends itself until the boss signals the thread that more work is available in the queue. If all the worker threads are to share a single queue, then the threads can be programmed to only process certain types of requests. If the request in the queue is not of the type a particular thread is to process, the thread can again suspend itself. The primary purpose of the boss thread is to create all the threads, place work in the queue, and awaken worker threads when work is available. The worker threads check the request in the queue, perform the assigned task, and suspend itself if no work is available. All the worker threads and the boss thread execute concurrently. [Figure 4-6](#) contrasts these two approaches for the delegation model.



Figure 4-6. The two approaches to the delegation model.

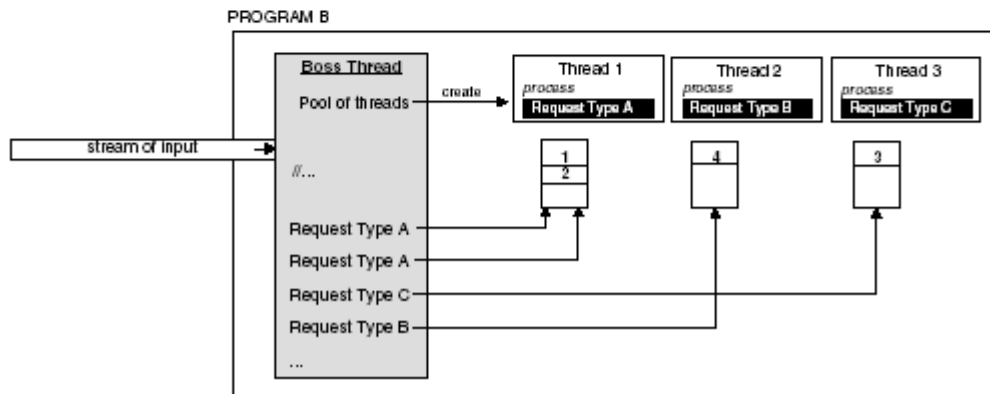
DELEGATION APPROACH 1:

Boss thread creates a new thread for each new request.



DELEGATION APPROACH 2:

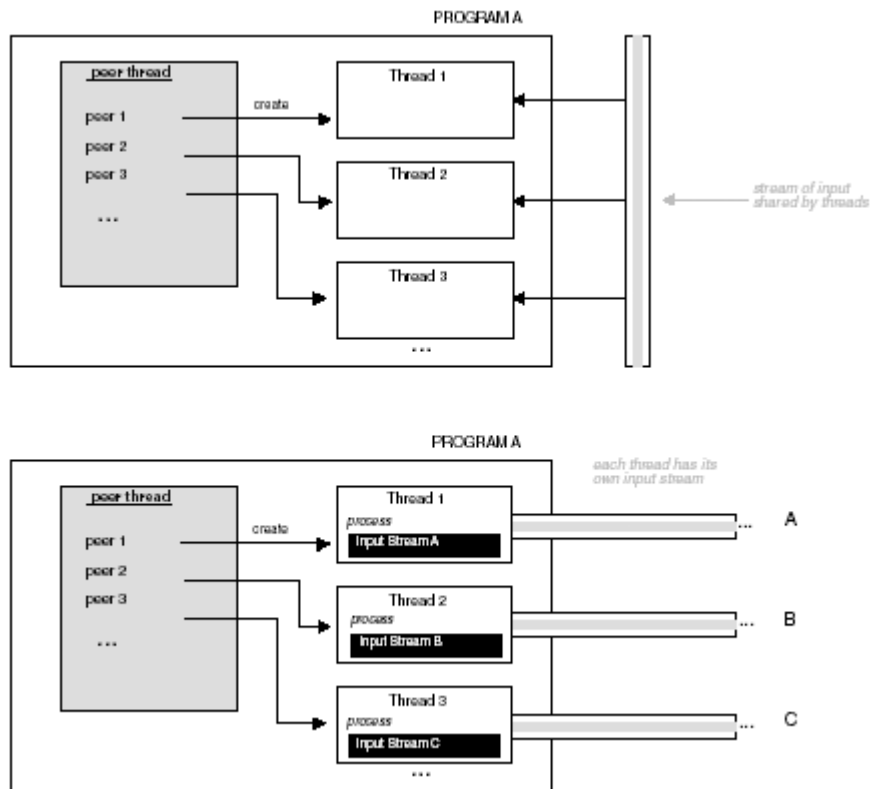
Boss thread creates a pool of threads that processes all requests.



### 4.5.2 Peer-to-Peer Model

Where the delegation model has a boss thread that delegates tasks to worker threads, in the peer-to-peer model all the threads have an equal working status. Although there is a single thread that initially creates all the threads needed to perform all the tasks, that thread is considered a worker thread and does no delegation. In this model, there is no centralized thread. The worker (peer) threads have more responsibility. The peer threads can process requests from a single input stream shared by all the threads or each thread may have its own input stream for which it is responsible. The input can also be stored in a file or database. The peer threads may have to communicate and share resources. [Figure 4-7](#) shows the peer-to-peer thread model.

Figure 4-7. Peer-to-peer thread model.

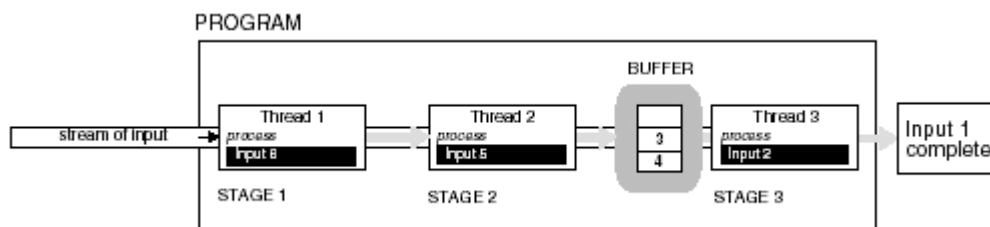


### 4.5.3 Pipeline Model

The pipeline model is characterized as an assembly line in which a stream of items are processed in stages. At each stage, work is performed on a unit of input by a thread. When the unit of input has been through all the stages, then the processing of the input has been completed. This approach allows multiple inputs to be processed simultaneously. Each thread is responsible for producing its interim results or output, making them available to the next stage or next thread in the pipeline. The last stage or thread produces the result of the pipeline.

As the input moves down the pipeline, it may be necessary to buffer units of input at certain stages as threads process previous input. This may cause a slowdown in the pipeline if a particular stage's processing is slower than other stages, causing a backlog. To prevent backlog, it may be necessary for that stage to create additional threads to process incoming input. The stages of work in a pipeline should be balanced where one stage does not take more time than the other stages. Work should be evenly distributed throughout the pipeline. More stages and therefore more threads may also be added to the pipeline. This will also prevent backlog. [Figure 4-8](#) shows the pipeline model.

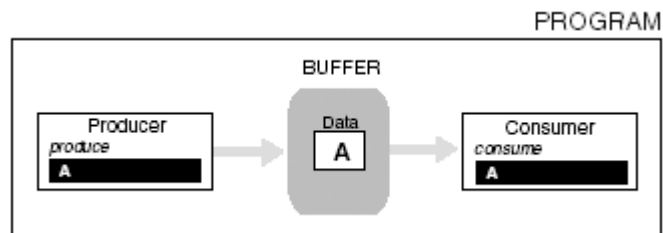
Figure 4-8. The pipeline model.



#### 4.5.4 Producer-Consumer Model

In the producer-consumer model, there is a producer thread that produces data to be consumed by the consumer thread. The data is stored in a block of memory shared between the producer and consumer threads. The producer thread must first produce data, then the consumer threads retrieve it. This process will require synchronization. If the producer thread deposits data at a much faster rate than the consumer thread consumes it, then the producer thread may at several times overwrite previous results before the consumer thread retrieves it. On the other hand, if the consumer thread retrieves data at a much faster rate than the producer deposits data, then the consumer thread may retrieve identical data or attempt to retrieve data not yet deposited. [Figure 4-9](#) shows the producer-consumer model. The producer-consumer model is also called the client-server model for larger-scaled programs and applications.

**Figure 4-9. The producer–consumer model.**

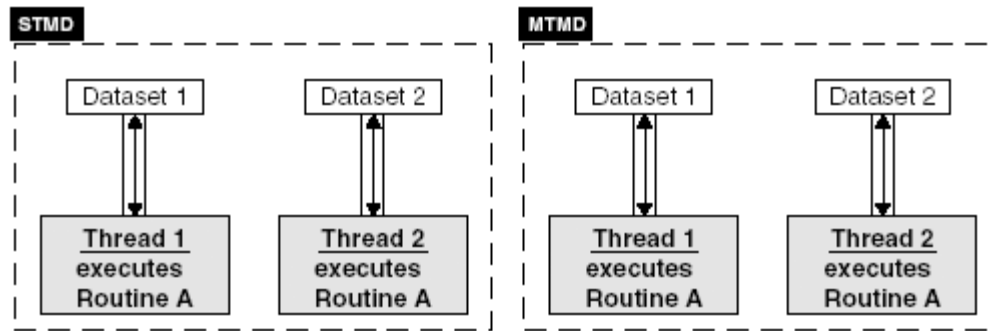


#### 4.5.5 SPMD and MPMD for Threads

In each of the previous thread models, the threads are performing the same task over and over again on different data sets or are assigned different tasks performed on different data sets. These thread models utilize SIMD (Single Instruction Multiple Data) or MPMD (Multiple Programs Multiple Data). These are two models of parallelism that classify programs by instruction and data streams. They can be used to describe the type of work the thread models are implementing in parallel. For our purposes, MPMD is better stated as MTMD (Multiple Threads Multiple Data). These models describe a system that executes different threads processing different sets of data or data streams. SPMD means Single Program Multiple Data or, for our purposes, STMD (Single Thread Multiple Data). This model describes a system that executes a single thread that processes different sets of data or data streams. This means several identical threads executing the same routine are given different sets of data to process.

The delegation and peer-to-peer models can both use STMD or MTMD models of parallelism. As described, the pool of threads can execute different routines processing different sets of data. This utilizes the MTMD model. The pool of threads can also be given the same routine to execute. The requests or jobs submitted to the system could be different sets of data instead of different tasks. In this case, a set of threads implementing the same instructions but on different sets of data thus utilizes STMD. The peer-to-peer model can be threads executing the same or different tasks. Each thread can have its own data stream or several files of data that each thread is to process. The pipeline model uses the MTMD model of parallelism. At each stage different processing is performed so multiple input units are at different stages of completion. The pipeline metaphor would be useless if at each stage the same processing was performed. [Figure 4-10](#) contrasts the STMD and MTMD models of parallelism.

Figure 4-10. The STMD and MTMD models of parallelism.



## 4.6 Introduction to the Pthread Library

The Pthread library supplies the API to create and manage the threads in your application. The Pthread library is based on a standardized programming interface for the creation and maintenance of threads. The thread interface has been specified by the IEEE standards committee in the POSIX 1003.1c standard. Third-party vendors supply an implementation that adheres to the POSIX standard. Their implementation is referred to as Pthreads or POSIX thread library.

The Pthread library contains over 60 functions that can be classified into the following categories:

- I. Thread Management Functions
  - I. Thread configuration
  - II. Thread cancellation
  - III. Thread scheduling
  - IV. Thread specific data
  - V. Signals
  - VI. Thread attribute functions
    - a. Thread attribute configuration
    - b. Thread attribute stack configuration
    - c. Thread attribute scheduling configuration
- II. Mutex Functions
  - I. Mutex configuration
  - II. Priority management
  - III. Mutex attribute functions
    - a. Mutex attribute configuration
    - b. Mutex attribute protocol
    - c. Mutex attribute priority management
- III. Condition Variable Functions
  - I. Condition variable configuration
  - II. Condition variable attribute functions
    - a. Condition variable attribute configuration

## b. Condition variable sharing functions

The Pthread library can be implemented in any language but in order to be compliant with the POSIX standard, they must comply to the standardized interface and behave in the manner specified. The Pthread library is not the only thread API implementation. Hardware and third-party vendors have implemented their own proprietary thread APIs. For example, the Sun environment supports the Pthread library and their own Solaris thread library. In this chapter, we discuss some Pthread functions that implement thread management.

## 4.7 The Anatomy of a Simple Threaded Program

Any simple multithreaded program will consist of a main or creator thread and the functions that the threads will execute. The thread models determine the manner in which the threads are created and managed. They can be created all at once or under certain conditions. In [Example 4.1](#) the delegation model is used to show a simple multithreaded program.

**Example 4.1 Using the delegation model in a simple threaded program.**

```
#include <iostream>
#include <pthread.h>

void *task1(void *X) //define task to be executed by ThreadA
{
    //...
    cout << "Thread A complete" << endl;
}

void *task2(void *X) //define task to be executed by ThreadB
{
    //...
    cout << "Thread B complete" << endl;
}

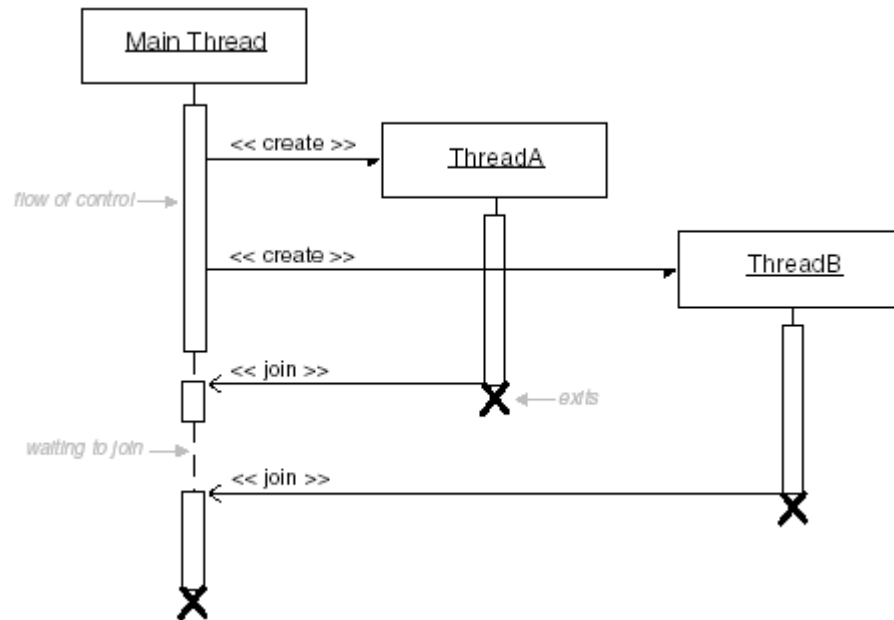
int main(int argc, char *argv[])
{
    pthread_t ThreadA,ThreadB; // declare threads

    pthread_create(&ThreadA,NULL,task1,NULL); // create threads
    pthread_create(&ThreadB,NULL,task2,NULL);
    // additional processing
    pthread_join(ThreadA,NULL); // wait for threads
    pthread_join(ThreadB,NULL);
    return(0);
}
```

In [Example 4.1](#), the main line of the example defines the set of instructions for the primary thread. The primary thread, in this case, is also the boss thread. The boss thread declares two threads, ThreadA and ThreadB. By using the `pthread_create()` function, these two threads are associated with the tasks they are to execute. The two tasks, `task1` and `task2`, are defined. They simply send a message to the standard out but could be programmed to do anything. The `pthread_create()` function causes the threads to immediately execute their assigned tasks. The `pthread_join()` function works the same way as `wait()` for processes. The primary thread waits until both threads return. [Figure 4-11](#) shows the layout of [Example 4.1](#). It also shows what happens to the flow of controls as the `pthread_create()` and `pthread_join()`

functions are called.

**Figure 4-11. The layout, output, and flow of control for [Example 4.1](#).**



In [Figure 4-11](#), the `pthread_create()` function causes a fork in the flow of control in the main line or primary thread. Two additional flows of control, one for each thread, are executing concurrently. The `pthread_create()` function returns immediately after the threads are created. It is an asynchronous function. As each thread executes its set of instructions, the primary thread executes its instructions. The `pthread_join()` causes the primary thread to wait until each thread terminates and rejoins the main flow of control.

#### 4.7.1 Compiling and Linking Multithreaded Programs

All multithreaded programs using the POSIX thread library must include the header:

```
<pthread.h>
```

In order to compile and link multithreaded applications in the UNIX or Linux environments using the `g++` or `gcc` command-line compilers, be sure to link the Pthreads library to your application. Use the `-l` option that specifies a library.

```
-lpthread
```

will cause your application to link to the library that is compliant with the multithreading interface defined by POSIX 1003.1c standard. The Pthread library, `libpthread.so`, should be located in the directory where the system stores its standard library, usually `/usr/lib`. If it is not located in a standard location, use the `-L` option to make the compiler look in a particular directory first before searching the standard locations.

```
g++ -o blackboard -L /src/local/lib blackboard.cpp -lpthread
```

tells the compiler to look in the `/src/local/lib` directory for the Pthread library before searching in the standard locations.

The complete programs in this book are accompanied by a program profile. The program profile contains implementation specifics such as headers and libraries required and compile and link

instructions. The profile also includes a note section that will contain any special considerations that need to be taken when executing the program.

## 4.8 Creating Threads

The Pthreads library can be used to create, maintain, and manage the threads of multithreaded programs and applications. When creating a multithreaded program, threads can be created any time during the execution of a process because they are dynamic. The `pthread_create()` function creates a new thread in the address space of a process. The thread parameter points to a thread handle or thread id of the thread that will be created. The new thread will have the attributes specified by the attribute object `attr`. The thread parameter will immediately execute the instructions in `start_routine` with the arguments specified by `arg`. If the function successfully creates the thread, it will return the thread id and store the value in the thread parameter.

If `attr` is `NULL`, the default thread attributes will be used by the new thread. The new thread takes on the attributes of `attr` when it is created. If `attr` is changed after the thread has been created, it will not affect any of the thread's attributes. If `start_routine` returns, the thread returns as if `pthread_exit()` had been called using the return value of `start_routine` as its exit status.

### Synopsis

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
                  const pthread_attr_t *restrict attr,
                  void *(*start_routine)(void*),
                  void *restrict arg);
```

If successful, the function will return 0. If the function is not successful, no new thread is created and the function will return an error number. If the system does not have the resources to create the thread or the thread limit for the process has been reached, the function will fail. The function will also fail if the thread attribute is invalid or the caller thread does not have permission to set the necessary thread attributes.

These are examples of creating two threads with default attributes:

```
pthread_create(&threadA, NULL, task1, NULL);
pthread_create(&threadB, NULL, task2, NULL);
```

These are the two `pthread_create()` function calls from [Example 4.1](#). Both threads are created with default attributes.

[Program 4.1](#) shows a primary thread passing an argument from the command line to the functions executed by the threads.

#### Program 4.1

```
#include <iostream>
#include <pthread.h>
#include <stdlib.h>
```

```

int main(int argc, char *argv[])
{
    pthread_t ThreadA,ThreadB;
    int N;

    if(argc != 2){
        cout << "error" << endl;
        exit (1);
    }

    N = atoi(argv[1]);
    pthread_create(&ThreadA,NULL,task1,&N);
    pthread_create(&ThreadB,NULL,task2,&N);
    cout << "waiting for threads to join" << endl;
    pthread_join(ThreadA,NULL);
    pthread_join(ThreadB,NULL);
    return(0);
}

```

[Program 4.1](#) shows how the primary thread can pass arguments from the command line to each of the thread functions. A number is typed in at the command line. The primary thread converts the argument to an integer and passes it to each function as a pointer to an integer as the last argument to the `pthread_create()` functions. [Program 4.2](#) shows each of the thread functions.

#### Program 4.2

```

void *task1(void *X)
{
    int *Temp;
    Temp = static_cast<int *>(X);

    for(int Count = 1;Count < *Temp;Count++){
        cout << "work from thread A: " << Count << " * 2 = "
            << Count * 2 << endl;
    }
    cout << "Thread A complete" << endl;
}

void *task2(void *X)
{
    int *Temp;
    Temp = static_cast<int *>(X);

    for(int Count = 1;Count < *Temp;Count++){
        cout << "work from thread B: " << Count << " + 2 = "
            << Count + 2 << endl;
    }
    cout << "Thread B complete" << endl;
}

```

In [Program 4.2](#), `task1` and `task2` executes a loop that is iterated the number of times as the value passed to the function. The function either adds or multiplies the loop invariant by 2 and sends the results to standard out. Once complete, each function outputs a message that the thread is complete. The instructions for compiling and executing [Programs 4.1](#) and [4.2](#) are contained in [Program Profile 4.1](#).



## Program Profile 4.1

Program Name

program4-12.cc

Description

Accepts an integer from the command line and passes the value to the thread functions. Each function executes a loop that either adds or multiplies the loop invariant by 2 and sends the result to standard out. The main line or primary thread is listed in [Program 4.1](#) and the functions are listed in [Program 4.2](#).

Libraries Required

libpthread

Headers Required

<pthread.h> <iostream> <stdlib.h>

Compile and Link Instructions

```
c++ -o program4-12 program4-12.cc -lpthread
```

Test Environment

SuSE Linux 7.1, gcc 2.95.2,

Execution Instructions

```
./program4-12 34
```

Notes

This program requires a command-line argument.

This is an example of passing a single argument to the thread function. If it is necessary to pass multiple arguments to the thread function, create a struct or container containing all the required arguments and pass a pointer to that structure to the thread function.

### 4.8.1 Getting the Thread Id

As mentioned earlier, the process shares all its resources with the threads in its address space. Threads have very few resources of their own. The thread id is one of the resources unique to each thread. The `pthread_self()` function returns the thread id of the calling thread.

## Synopsis

```
#include <pthread.h>

pthread_t pthread_self(void);
```

This function is similar to `getpid()` for processes. When a thread is created, the thread id is returned to the creator or calling thread. The thread id will not know the created thread. Once the thread has its own id, it can be passed to other threads in the process. This function returns the thread id with no errors defined.

Here is an example of calling this function:

```
//...
pthread_t ThreadId;
ThreadId = pthread_self();
```

A thread calls this function and the function returns the thread id stored in the variable `ThreadId` of type `pthread_t`.

### 4.8.2 Joining Threads

The `pthread_join()` function is used to join or rejoin flows of control in a process. The `pthread_join()` causes the calling thread to suspend its execution until the target thread has terminated. It is similar to the `wait()` function used by processes. This function can be called by the creator of a thread. The creator thread waits for the new thread to terminate and return, thus rejoining flows of control. The `pthread_join()` can also be called by peer threads if the thread handle is global. This will allow any thread to join flows of control with any other thread in the process. If the calling thread is canceled before the target thread returns, the target thread will not become a detached thread (discussed in the next section). If different peer threads simultaneously call the `pthread_join()` function on the same thread, this behavior is undefined.

## Synopsis

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

The `thread` parameter is the thread (target thread) the calling thread is waiting on. If the function returns successfully, the exit status is stored in `value_ptr`. The exit status is the argument passed to the `pthread_exit()` function called by the terminated thread. The function will return an error number if it fails. The function will fail if the target thread is not a joinable thread or, in other words, created as a detached thread. The function will also fail if the specified thread does not exist.

There should be a `pthread_join()` function called for all joinable threads. Once the thread is joined, this will allow the operating system to reclaim storage used by the thread. If a joinable thread is not joined to any thread or the thread that calls the join function is canceled, then the target thread will continue to utilize storage. This is a state similar to a zombied process when the parent process has not accepted the exit status of a child process, the child process continues to occupy an entry in the process table.

### 4.8.3 Creating Detached Threads

A detached thread is a terminated thread that is not joined or waited upon by any other threads. When the thread terminates, the limited resources used by the thread, including the thread id, are reclaimed and returned to the system pool. There is no exit status for any thread to obtain. Any thread that attempts to call `pthread_join()` for a detached thread will fail. The `pthread_detach()` function detaches the thread specified by `thread`. By default, all threads are created as joinable unless otherwise specified by the thread attribute object. This function detaches already existing joinable threads. If the thread has not terminated, a call to this function does not cause it to terminate.

#### Synopsis

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

If successful, the function will return 0. If not successful, it will return an error number. The `pthread_detach()` function will fail if `thread` is already detached or the thread specified by `thread` could not be found.

This is an example of detaching an already existing joinable thread:

```
//...
pthread_create(&threadA, NULL, task1, NULL);
pthread_detach(threadA);
//...
```

This causes `threadA` to be a detached thread. To create a detached thread, as opposed to dynamically detaching a thread, requires setting the detachstate of a thread attribute object and using that attribute object when the thread is created.

### 4.8.4 Using the Pthread Attribute Object

The thread attribute object encapsulates the attributes of a thread or group of threads. It is used to set the attributes of threads during their creation. The thread attribute object is of type `pthread_attr_t`. This structure can be used to set these thread attributes:

- size of the thread's stack
- location of the thread's stack
- scheduling inheritance, policy, and parameters
- whether the thread is detached or joinable
- the scope of the thread

The `pthread_attr_t` has several methods that can be invoked to set and retrieve each of these attributes. [Table 4-3](#) lists the methods used to set the attributes of the attribute object.

The `pthread_attr_init()` and `pthread_attr_destroy()` functions are used to initialize and destroy a thread attribute object.

## Synopsis

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

The `pthread_attr_init()` function initializes a thread attribute object with the default values for all the attributes. The `attr` parameter is a pointer to a `pthread_attr_t` object. Once `attr` has been initialized, its attribute values can be changed by using the `pthread_attr_set` functions listed in [Table 4-3](#). Once the attributes have been appropriately modified, `attr` can be used as a parameter in any call to the `pthread_create()` function. If successful, the function will return 0. If not successful, the function will return an error number. The `pthread_attr_init()` function will fail if there is not enough memory to create the object.

The `pthread_attr_destroy()` function can be used to destroy a `pthread_attr_t` object specified by `attr`. A call to this function deletes any hidden storage associated with the thread attribute object. If successful, the function will return 0. If not successful, the function will return an error number.

### 4.8.4.1 Creating Detached Threads Using the Pthread Attribute Object

Once the thread object has been initialized, its attributes can be modified. The `pthread_attr_setdetachstate()` function can be used to set the `detachstate` attribute of the attribute object. The `detachstate` parameter describes the thread as detached or joinable.

## Synopsis

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr,
                               int *detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                               int *detachstate);
```

The `detachstate` can have one of these values:

```
PTHREAD_CREATE_DETACHED
PTHREAD_CREATE_JOINABLE
```

The `PTHREAD_CREATE_DETACHED` value will cause all the threads that use this attribute object to be detached. The `PTHREAD_CREATE_JOINABLE` value will cause all the threads that use this attribute object to be joinable. This is the default value of `detachstate`. If successful, the function will return 0. If not successful, the function will return an error number. The `pthread_attr_setdetachstate()` function will fail if the value of `detachstate` is not valid.

The `pthread_attr_getdetachstate()` function will return the `detachstate` of the attribute object. If successful, the function will return the value of `detachstate` to the `detachstate` parameter and 0 as the return value. If not successful, the function will return an error number. In [Example 4.2](#), the threads created in [Program 4.1](#) are detached. This example uses an attribute object when creating one of the threads.

#### Example 4.2 Using an attribute object to create a detached thread.

```
//...

int main(int argc, char *argv[])
{
    pthread_t ThreadA,ThreadB;
    pthread_attr_t DetachedAttr;
    int N;

    if(argc != 2){
        cout << "error" << endl;
        exit (1);
    }

    N = atoi(argv[1]);
    pthread_attr_init(&DetachedAttr);
    pthread_attr_setdetachstate(&DetachedAttr,PTHREAD_CREATE_DETACHED);
    pthread_create(&ThreadA,NULL,task1,&N);
    pthread_create(&ThreadB,&DetachedAttr,task2,&N);
    cout << "waiting for thread A to join" << endl;
    pthread_join(ThreadA,NULL);
    return(0);
}
```

[Example 4.2](#) declares an attribute object `DetachedAttr`. The `pthread_attr_init()` function is used to allocate the attribute object. Once initialized, the `pthread_attr_detachstate()` function is used to change the detachstate from joinable to detached using the `PTHREAD_CREATE_DETACHED` value. When creating `ThreadB`, the `Detached-Attr` is the second argument in the call to the `pthread_create()` function. The `pthread_join()` call is removed for `ThreadB` because detached threads cannot be joined.

## 4.9 Managing Threads

When creating applications with multiple threads, there are several ways to control how threads perform and how threads use and compete for resources. Part of managing threads is setting the scheduling policy and priority of the threads. This contributes to the performance of the thread. Thread performance is also determined by how the threads compete for resources, either on a process or system scope. The scheduling, priority, and scope of the thread can be set by using a thread attribute object. Because threads share resources, access to resources will have to be synchronized. This will briefly be discussed in this chapter and fully discussed in [Chapter 5](#). Thread synchronization also includes when and how threads are terminated and canceled.

### 4.9.1 Terminating Threads

A thread's execution can be discontinued by several means:

- By returning from the execution of its assigned task with or without an exit status or return value
- By explicitly terminating itself and supplying an exit status
- By being canceled by another thread in the same address space

When a joinable thread function has completed executing, it returns to the thread calling `pthread_join()`, for which it is the target thread. The `pthread_join()` returns the exit status passed to the `pthread_exit()`

function called by the terminating thread. If the terminating thread did not make a call to `pthread_exit()`, then the exit status will be the return value of the function, if it has one; otherwise, the exit status is `NULL`.

It may be necessary for one thread to terminate another thread in the same process. For example, an application may have a thread that monitors the work of other threads. If a thread performs poorly or is no longer needed, to save system resources it may be necessary to terminate that thread. The terminating thread may terminate immediately or defer termination until a logical point in its execution. The terminating thread may also have to perform some cleanup tasks before it terminates. The thread also has the option to refuse termination.

The `pthread_exit()` function is used to terminate the calling thread. The `value_ptr` is passed to the thread that calls `pthread_join()` for this thread. Cancellation cleanup handler tasks that have not executed will execute along with the destructors for any thread-specific data. No resources used by the thread are released.

## Synopsis

```
#include <pthread.h>

int pthread_exit(void *value_ptr);
```

When the last thread of a process exits, then the process has terminated with an exit status of 0. This function cannot return to the calling thread and there are no errors defined.

The `pthread_cancel()` function is used to cancel the execution of another thread in the same address space. The thread parameter is the thread to be canceled.

## Synopsis

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

A call to the `pthread_cancel()` function is a request to cancel a thread. The request can be granted immediately, at a later time, or ignored. The cancel type and cancel state of the target thread determines when or if thread cancellation actually takes place. When the request is granted, there is a cancellation process that occurs asynchronously to the returning of the `pthread_cancel()` function to the calling thread. If the thread has cancellation cleanup handler tasks, they are performed. When the last handler returns, the destructors for thread-specific data, if any, are called and the thread is terminated. This is the cancellation process. The function returns 0 if successful and an error if not successful. The `pthread_cancel()` function will fail if the thread parameter does not correspond to an existing thread.

Some threads may require safeguards against untimely cancellation. Installing safeguards in a thread's function may prevent undesirable situations. Threads share data and depending on the thread model used, one thread may be processing data that is to be passed to another thread for processing. While the thread is processing data, it has sole possession by locking a mutex associated with the data. If a thread has locked a mutex and is canceled before the mutex is released, this could cause deadlock. The data may be required to be in some state before it can be used again. If a thread is canceled before this is done, an undesirable condition may occur. To put it simply, depending on the type of processing a thread is performing, thread cancellation should be performed when it is safe. A vital thread may prevent cancellation entirely. Therefore, thread cancellation should be restricted to threads that are not

vital or points of execution that do not have locks on resources. Cancellations can also be postponed until all vital cleanups have taken place.

The cancelability state describes the cancel condition of a thread as being cancelable or uncancelable. A thread's cancelability type determines the thread's ability to continue after a cancel request. A thread can act upon a cancel request immediately or defer the cancellation to a later point in its execution. The cancelability state and type are dynamically set by the thread itself.

The `pthread_setcancelstate()` and `pthread_setcanceltype()` functions are used to set the cancelability state and type of the calling thread. The `pthread_setcancelstate()` function sets the calling thread to the cancelability state specified by `state` and returns the previous state in `oldstate`.

## Synopsis

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

The values for `state` and `oldstate` are:

```
PTHREAD_CANCEL_DISABLE
PTHREAD_CANCEL_ENABLE
```

`PTHREAD_CANCEL_DISABLE` is a state in which a thread will ignore a cancel request. `PTHREAD_CANCEL_ENABLE` is a state in which a thread will concede to a cancel request. This is the default state of any newly created thread. If successful, the function will return 0. If not successful, the function will return an error number. The `pthread_setcancelstate()` may fail if not passed a valid state value.

The `pthread_setcanceltype()` function sets the calling thread to the cancelability type specified by `type` and returns the previous state in `oldtype`. The values for `type` and `oldtype` are:

```
PTHREAD_CANCEL_DEFFERED
PTHREAD_CANCEL_ASYNCHRONOUS
```

`PTHREAD_CANCEL_DEFFERED` is a cancelability type in which a thread puts off termination until it reaches its cancellation point. This is the default cancelability type for any newly created threads. `PTHREAD_CANCEL_ASYNCHRONOUS` is a cancelability type in which a thread terminates immediately. If successful, the function will return 0. If not successful, the function will return an error number. The `pthread_setcanceltype()` may fail if not passed a valid type value.

The `pthread_setcancelstate()` and `pthread_setcanceltype()` functions are used together to establish the cancelability of a thread. [Table 4-5](#) list combinations of state and type and a description of what will occur for each combination.

**Table 4-5. Combinations of Cancelability State and Type**

Cancelability State	Cancelability Type	Description
<code>PTHREAD_CANCEL_ENABLE</code>	<code>PTHREAD_CANCEL_DEFERRED</code>	Deferred cancellation. The default cancellation state and type of a thread. Thread cancellation takes places

Cancelability State	Cancelability Type	Description
		when it enters a cancellation point or when the programmer defines a cancellation point with a call to <code>pthread_testcancel()</code> .
<code>PTHREAD_CANCEL_ENABLE</code>	<code>PTHREAD_CANCEL_ASYNCHRONOUS</code>	Asynchronous cancellation. Thread cancellation takes place immediately.
<code>PTHREAD_CANCEL_DISABLE</code>	Ignored	Disabled cancellation. Thread cancellation does not take place.

#### 4.9.1.1 Cancellation Points

When a cancel request is deferred, the termination of the thread takes place later in the execution of the thread's function. Whenever it occurs, it should be "safe" to cancel the thread because it is not in the middle of executing critical code, locking a mutex, or leaving the data in some usable state. These safe locations in the code's execution are good locations for cancellation points. A cancellation point is a check point where a thread checks if there are any cancellation requests pending and, if so, concedes to termination.

Cancellation points can be marked by a call to `pthread_testcancel()`. This function checks for any pending cancellation request. If a request is pending, then it causes the cancellation process to occur at the location this function is called. If there are no cancellations pending, then the function continues to execute with no repercussions. This function call can be placed at any location in the code where it is considered safe to terminate the thread.

### Synopsis

```
#include <pthread.h>

void pthread_testcancel(void);
```

[Program 4.3](#) contains functions that use the `pthread_setcancelstate()`, `pthread_setcanceltype()`, and `pthread_testcancel()` functions. [Program 4.3](#) shows three functions setting their cancelability types and states.

#### Program 4.3

```
#include <iostream>
#include <pthread.h>

void *task1(void *X)
{
    int OldState;

    // disable cancelability
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &OldState);
```



```

    for(int Count = 1;Count < 100;Count++)
    {
        cout << "thread A is working: " << Count << endl;
    }
}

void *task2(void *X)
{
    int OldState,OldType;

    // enable cancelability, asynchronous
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,&OldState);
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,&OldType);

    for(int Count = 1;Count < 100;Count++)
    {
        cout << "thread B is working: " << Count << endl;
    }
}

void *task3(void *X)
{
    int OldState,OldType;

    // enable cancelability, deferred
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,&OldState);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,&OldType);

    for(int Count = 1;Count < 1000;Count++)
    {
        cout << "thread C is working: " << Count << endl;
        if((Count%100) == 0){
            pthread_testcancel();
        }
    }
}
}

```

In [Program 4.3](#), each task has set its cancelability condition. In task1, the cancelability of the thread has been disabled. What follows is critical code that must be executed. In task2, the cancelability of the thread is enabled. A call to the `pthread_setcancelstate()` is unnecessary because all new threads have an enabled cancelability state. The cancelability type is set to `PTHREAD_CANCEL_ASYNCHRONOUS`. This means whenever a cancel request is issued, the thread will start its cancellation process immediately, regardless of where it is in its execution. Therefore, it should not be executing any vital code once this type is activated. If it is making any system calls, they should be cancellation-safe functions (discussed later). In task2, the loop iterates until the cancel request is issued. In task3, the cancelability of the thread is also enabled and the cancellation type is `PTHREAD_CANCEL_DEFFERED`. This is the default state and type of a newly created thread, therefore, calls to the `pthread_setcancelstate()` and `pthread_setcanceltype()` are unnecessary. Critical code can be executed after the state and type are set because the termination will not take place until the `pthread_testcancel()` function is called. If there is no request pending, then the thread will continue executing until, if any, calls to `pthread_testcancel()` are made. In task3, the `pthread_cancel()` function is called whenever Count is evenly divisible by 100. Code between cancellation points should not be

critical because it may not execute.

[Program 4.4](#) shows the boss thread that issues the cancellation request for each thread.

#### Program 4.4

```
int main(int argc, char *argv[])
{
    pthread_t Threads[3];
    void *Status;

    pthread_create(&(Threads[0]),NULL,task1,NULL);
    pthread_create(&(Threads[1]),NULL,task2,NULL);
    pthread_create(&(Threads[2]),NULL,task3,NULL);

    pthread_cancel(Threads[0]);
    pthread_cancel(Threads[1]);
    pthread_cancel(Threads[2]);

    for(int Count = 0;Count < 3;Count++)
    {
        pthread_join(Threads[Count],&Status);

        if(Status == PTHREAD_CANCELED){
            cout << "thread" << Count << " has been canceled" << endl;
        }
        else{
            cout << "thread" << Count << " has survived" << endl;
        }
    }

    return(0);
}
```

The boss thread in [Program 4.4](#) creates three threads, then it issues a cancellation request for each thread. The boss thread calls the `pthread_join()` function for each thread. The `pthread_join()` function does not fail if it attempts to join with a thread that has already been terminated. The join function just retrieves the exit status of the terminated thread. This is good because the thread that issues the cancellation request may be a different thread than the thread that calls `pthread_join()`. Monitoring the work of all the worker threads may be the sole task of a single thread that also cancels threads. Another thread may examine the exit status of threads by calling the `pthread_join()` function. This type of information may be used to statistically evaluate which threads have the best performance. In this program, the boss thread joins and examines each exit thread's exit status in a loop. Thread[0] was not canceled because its cancelability was disabled. The other two threads were canceled. A canceled thread may return an exit status, for example, `PTHREAD_CANCELED`. [Program Profile 4.2](#) contains the profile for [Programs 4.3](#) and [4.4](#).

## Program Profile 4.2

Program Name

program4-34.cc

Description

Demonstrates the use of thread cancellation. Three threads have different cancellation types and states. Each thread executes a loop. The cancellation state and type determines the number of loop iterations or whether the loop is executed at all. The primary thread examines the exit status of each thread.

#### Libraries Required

libpthread

#### Headers Required

<pthread.h> <iostream>

#### Compile and Link Instructions

```
c++ -o program4-34 program4-34.cc -lpthread
```

#### Test Environment

SuSE Linux 7.1, gcc 2.95.2,

#### Execution Instructions

```
./program4-34
```

Cancellation points marked by a call to the `pthread_testcancel()` function are used in user-defined functions. The Pthread library defines the execution of other functions as cancellation points. These functions block the calling thread and while blocked the thread is safe to be canceled. These are the Pthread library functions that act as cancellation points:

```
pthread_testcancel()  
pthread_cond_wait()  
pthread_timedwait  
pthread_join()
```

If a thread with a deferred cancelability state has a cancellation request pending when making a call to one of these Pthread library functions, the cancellation process will be initiated. As far as system calls, [Table 4-6](#) lists some of the system calls required to be cancellation points.

While these functions are safe for deferred cancellation, they may not be safe for asynchronous cancellation. An asynchronous cancellation during a library call that is not an asynchronously safe function may cause library data to be left in an incompatible state. The library may have allocated memory on behalf of the thread and when the thread is canceled, may still have a hold on that memory. For other library and systems functions that are not cancellation safe (asynchronously or deferred), it may be necessary to write code preventing a thread from terminating by disabling cancellation or deferring cancellation until after the function call has returned.

#### 4.9.1.2 Cleaning Up Before Termination

Once the thread concedes to cancellation, it may need to perform some final processing before it is terminated. The thread may have to close files, reset shared resources to some consistent state, release locks, or deallocate resources. The Pthread library defines a mechanism for each thread to perform last-

minute tasks before terminating. A cleanup stack is associated with every thread. The stack contains pointers to routines that are to be executed during the cancellation process. The `pthread_cleanup_push()` function pushes a pointer to the routine onto the cleanup stack.

**Table 4-6. POSIX System Calls Required to be Cancellation Points**

**POSIX System Calls (Cancellation Points)**

<code>accept()</code>	<code>nanosleep()</code>	<code>sem_wait()</code>
<code>aio_suspend()</code>	<code>open()</code>	<code>send()</code>
<code>clock_nanosleep()</code>	<code>pause()</code>	<code>sendmsg()</code>
<code>close()</code>	<code>poll()</code>	<code>sendto()</code>
<code>connect()</code>	<code>pread()</code>	<code>sigpause()</code>
<code>creat()</code>	<code>pthread_cond_timedwait()</code>	<code>sigsuspend()</code>
<code>fcntl()</code>	<code>pthread_cond_wait()</code>	<code>sigtimedwait()</code>
<code>fsync()</code>	<code>pthread_join()</code>	<code>sigwait()</code>
<code>getmsg()</code>	<code>putmsg()</code>	<code>sigwaitinfo()</code>
<code>lockf()</code>	<code>putpmsg()</code>	<code>sleep()</code>
<code>mq_receive()</code>	<code>pwrite()</code>	<code>system()</code>
<code>mq_send()</code>	<code>read()</code>	<code>usleep()</code>
<code>mq_timedreceive()</code>	<code>readv()</code>	<code>wait()</code>
<code>mq_timedsend()</code>	<code>recvfrom()</code>	<code>waitpid()</code>
<code>msgrcv()</code>	<code>recvmsg()</code>	<code>write()</code>
<code>msgsnd()</code>	<code>select()</code>	<code>writew()</code>

## POSIX System Calls (Cancellation Points)

msync()

sem\_timedwait()

### Synopsis

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

The routine parameter is a pointer to the function to be pushed onto the stack. The arg parameter is passed to the function. The function routine is called with the arg parameter when the thread exits by calling `pthread_exit()`, when the thread concedes to a termination request, or when the thread explicitly calls the `pthread_cleanup_pop()` function with a nonzero value for `execute`. The function does not return.

The `pthread_cleanup_pop()` function removes routine's pointer from the top of the calling thread's cleanup stack. The `execute` parameter can have a value of 1 or 0. If the value is 1, the thread executes routine even if it is not being terminated. The thread continues execution from the point after the call to this function. If the value is 0, the pointer is removed from the top of the stack without executing.

It is required for each push there be a pop within the same lexical scope. For example, `funcA()` requires a cleanup handler to be executed when the function exits or cancels:

```
void *funcA(void *X)
{
    int *Tid;
    Tid = new int;
    // do some work
    //...
    pthread_cleanup_push(cleanup_funcA,Tid);
    // do some more work
    //...
    pthread_cleanup_pop(0);
}
```

Here, `funcA()` pushes cleanup handler `cleanup_funcA()` onto the cleanup stack by calling the `pthread_cleanup_push()` function. The `pthread_cleanup_pop()` function is required for each call to the `pthread_cleanup_push()` function. The pop function is passed 0, which means the handler is removed from the cleanup stack but is not executed at this point. The handler will be executed if the thread that executes `funcA()` is canceled.

The `funcB()` also requires a cleanup handler:

```
void *funcB(void *X)
{
    int *Tid;
    Tid = new int;
    // do some work
    //...
    pthread_cleanup_push(cleanup_funcB,Tid);

    // do some more work
    //...
```

```

    pthread_cleanup_pop(1);
}

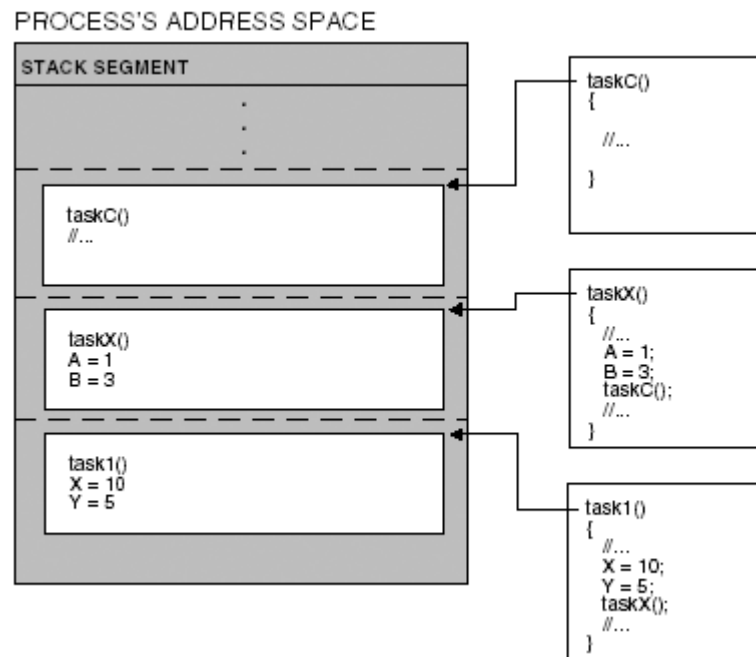
```

Here, funcB() pushes cleanup handler cleanup\_funcB() onto the cleanup stack. The difference in this case is the pthread\_cleanup\_pop() function is passed 1, which means the handler is removed from the cleanup stack but will execute at this point. The handler will be executed regardless of whether the thread that executes funcA() is canceled or not. The cleanup handlers, cleanup\_funcA() and cleanup\_funcB(), are regular functions that can be used to close files, release resources, unlock mutexes, and so on.

## 4.9.2 Managing the Thread's Stack

The address space of a process is divided into the text and static data segments, free store, and the stack segment. The location and size of the thread's stacks are cut out of the stack segment of the process. A thread's stack will store a stack frame for each routine it has called but has not exited. The stack frame contains temporary variables, local variables, return addresses, and any other additional information the thread needs to find its way back to previously executing routines. Once the routine is exited, the stack frame for that routine is removed from the stack. [Figure 4-12](#) shows how stack frames are placed onto a stack.

**Figure 4-12. Stack frames generated from a thread.**



In [Figure 4-12](#), Thread A executes Task 1. Task 1 creates some local variables, does some processing, then calls Task X. A stack frame is created for Task 1 and placed on the stack. Task X does some processing, creates local variables, then calls Task C. A stack frame for Task X is placed on the stack. Task C calls Task Y, and so on. Each stack must be large enough to accommodate the execution of each thread's function along with the chain of routines that will be called. The size and location of a thread's stack are managed by the operating system but they can be set or examined by several methods defined by the attribute object.

The pthread\_attr\_getstacksize() function returns the default stack size minimum. The attr parameter is the thread attribute object from which the default stack size is extracted. When the function returns, the default stack size, expressed in bytes, is stored in the stacksize parameter and the return value is 0. If

not successful, the function returns an error number.

The `pthread_attr_setstacksize()` function sets the stack size minimum. The `attr` parameter is the thread attribute object for which the stack size is set. The `stacksize` parameter is the minimum size of the stack expressed in bytes. If the function is successful, the return value is 0. If not successful, the function returns an error number. The function will fail if `stacksize` is less than `PTHREAD_MIN_STACK` or exceeds the system minimum. The `PTHREAD_STACK_MIN` will probably be a lower minimum than the default stack minimum returned by `pthread_attr_getstacksize()`. Consider the value returned by the `pthread_attr_getstacksize()` before raising the minimum size of a thread's stack. A stack's size is fixed so the stack's growth during runtime will only be within the fixed space of the stack set at compile time.

## Synopsis

[\[View full width\]](#)

```
#include <pthread.h>

void pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                              void **restrict stacksize);
void pthread_attr_setstacksize(pthread_attr_t *attr, void
➔ *stacksize);
```

The location of the thread's stack can be set and retrieved by the `pthread_attr_setstackaddr()` and `pthread_attr_getstackaddr()` functions. The `pthread_attr_setstackaddr()` function sets the base location of the stack to the address specified by the parameter `stackattr` for the thread created with the thread attribute object `attr`. This address `addr` should be within the virtual address space of the process. The size of the stack will be at least equal to the minimum stack size specified by `PTHREAD_STACK_MIN`. If successful, the function will return 0. If not successful, the function will return an error number.

The `pthread_attr_getstackaddr()` function retrieves the base location of the stack address for the thread created with the thread attribute object specified by the parameter `attr`. The address is returned and stored in the parameter `stackaddr`. If successful, the function will return 0. If not successful, the function will return an error number.

## Synopsis

[\[View full width\]](#)

```
#include <pthread.h>

void pthread_attr_setstackaddr(pthread_attr_t *attr, void
➔ *stackaddr);
void pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
                              void **restrict stackaddr);
```

The stack attributes (size and location) can be set by a single function. The `pthread_attr_setstack()` function sets both the stack size and stack location of a thread created using the specified attribute object `attr`. The base location of the stack will be set to the `stackaddr` parameter and the size of the stack will be set to the `stacksize` parameter. The `pthread_attr_getstack()` function retrieves the stack size and stack location of a thread created using the specified attribute object `attr`. If successful, the stack location will be stored in the `stackaddr` parameter and the stack size will be stored in the `stacksize`

parameter. If successful, these functions will return 0. If not successful, an error number is returned. The `pthread_setstack()` function will fail if the `stacksize` is less than `PTHREAD_STACK_MIN` or exceeds some implementation-defined limit.

## Synopsis

[\[View full width\]](#)

```
#include <pthread.h>

void pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
                           size_t stacksize);
void pthread_attr_getstack(const pthread_attr_t *restrict attr,
                           void **restrict stackaddr, size_t
➔ stacksize);
```

[Example 4.3](#) sets the stack size of a thread using a thread attribute object.

**Example 4.3 Changing the stack size of a thread using an offset.**

```
//...

pthread_attr_getstacksize(&SchedAttr,&DefaultSize);
if(DefaultSize < Min_Stack_Req){

    SizeOffset = Min_Stack_Req - DefaultSize;
    NewSize = DefaultSize + SizeOffset;
    pthread_attr_setstacksize(&Attr1,(size_t)NewSize);
}
```

In [Example 4.3](#), the thread attribute object retrieves the default size from the attribute object then determines whether the default size is less than the minimum stack size desired. If so, the offset is calculated then added to the default stack size. This becomes the new minimum stack size for this thread.

NOTE:

Setting the stack size and stack location may cause your program to be nonportable. The stack size and location you set for your program on one platform may not match the stack size and location of another platform.

### 4.9.3 Setting Thread Scheduling and Priorities

Like processes, threads execute independently. Each thread is assigned to a processor in order to execute the task it has been given. Each thread is assigned a scheduling policy and priority that dictates how and when it is assigned to a processor. The scheduling policy and priority of a thread or group of threads can be set by an attribute object using these functions:

```
pthread_attr_setinheritsched()
pthread_attr_setschedpolicy()
pthread_attr_setschedparam()
```

These functions can be used to return scheduling information about the thread:



```
pthread_attr_getinheritsched()
pthread_attr_getschedpolicy()
pthread_attr_getschedparam()
```

## Synopsis

```
#include <pthread.h>
#include <sched.h>

void pthread_attr_setinheritsched(pthread_attr_t *attr,
                                  int inheritsched);
void pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
void pthread_attr_setschedparam(pthread_attr_t *restrict
                                 attr, const struct sched_param
                                 *restrict param);
```

The `pthread_attr_setinheritsched()`, `pthread_attr_setschedpolicy()`, and `pthread_attr_setschedparam()` are used together to set the scheduling policy and priority of a thread. The `pthread_attr_setinheritsched()` function is used to determine how the thread's scheduling attributes will be set, either by inheriting the scheduling attributes from the creator thread or from an attribute object. The `inheritsched` parameter can have one of these values:

`PTHREAD_INHERIT_SCHED` Thread scheduling attributes shall be inherited from the creator thread and any scheduling attributes of the `attr` parameter will be ignored.

`PTHREAD_EXPLICIT_SCHED` Thread scheduling attributes shall be set to the scheduling attributes of the attribute object `attr`.

If the `inheritsched` parameter value is `PTHREAD_EXPLICIT_SCHED`, then the `pthread_attr_setschedpolicy()` function is used to set the scheduling policy and the `pthread_attr_setschedparam()` function is used to set the priority.

The `pthread_attr_setschedpolicy()` function sets the scheduling policy of the thread attribute object `attr`. The policy parameter values can be one of the following defined in the `<sched.h>` header:

`SCHED_FIFO` First-In-First-Out scheduling policy where the executing thread runs to completion.

`SCHED_RR` Round-robin scheduling policy where each thread is assigned to a processor only for a time slice.

`SCHED_OTHER` Other scheduling policy (implementation-defined). By default, this is the scheduling policy of any newly created thread.

The `pthread_attr_setschedparam()` function is used to set the scheduling parameters of the attribute object `attr` used by the scheduling policy. The `param` parameter is a structure that contains the parameters. The `sched_param` structure has at least this data member defined:

```
struct sched_param {
    int sched_priority;
    //...
};
```

It may also have additional data members along with several functions that return and set the priority minimum, maximum, scheduler, parameters, and so on. If the scheduling policy is either SCHED\_FIFO or SCHED\_RR, then the only member required to have a value is sched\_priority.

To obtain the maximum and minimum priority values, use the sched\_get\_priority\_min() and sched\_get\_priority\_max() functions.

## Synopsis

```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

Both functions are passed the scheduling policy policy for which the priority values are requested and both will return either the maximum or minimum priority values for the scheduling policy.

[Example 4.4](#) shows how to set the scheduling policy and priority of a thread by using the thread attribute object.

**Example 4.4 Using the thread attribute object to set the scheduling policy and priority of a thread.**

```
//...
#define Min_Stack_Req 3000000

pthread_t ThreadA;
pthread_attr_t SchedAttr;
size_t DefaultSize,SizeOffset,NewSize;
int MinPriority,MaxPriority,MidPriority;
sched_param SchedParam;

int main(int argc, char *argv[])
{
    //...
    // initialize attribute object
    pthread_attr_init(&SchedAttr);

    // retrieve min and max priority values for scheduling policy
    MinPriority = sched_get_priority_max(SCHED_RR);
    MaxPriority = sched_get_priority_min(SCHED_RR);

    // calculate priority value
    MidPriority = (MaxPriority + MinPriority)/2;

    // assign priority value to sched_param structure
    SchedParam.sched_priority = MidPriority;

    // set attribute object with scheduling parameter
    pthread_attr_setschedparam(&Attr1,&SchedParam);

    // set scheduling attributes to be determined by attribute object
    pthread_attr_setinheritsched(&Attr1,PTHREAD_EXPLICIT_SCHED);

    // set scheduling policy
    pthread_attr_setschedpolicy(&Attr1,SCHED_RR);
```

```

    // create thread with scheduling attribute object
    pthread_create(&ThreadA,&Attr1,task2,Value);
}

```

In [Example 4.4](#), the scheduling policy and priority of ThreadA is set using the thread attribute object SchedAttr. This is done in eight steps:

1. Initialize attribute object.
2. Retrieve min and max priority values for scheduling policy.
3. Calculate priority value.
4. Assign priority value to sched\_param structure.
5. Set attribute object with scheduling parameter.
6. Set scheduling attributes to be determined by attribute object.
7. Set scheduling policy.
8. Create thread with scheduling attribute object.

With this method, the scheduling policy and priority is set before the thread is running. In order to dynamically change the scheduling policy and priority, use the pthread\_setschedparam() and pthread\_setschedprio() functions.

**Synopsis**

```

#include <pthread.h>

int pthread_setschedparam(pthread_t thread, int policy,
                          const struct sched_param *param);
int pthread_getschedparam(pthread_t thread, int *restrict policy,
                          struct sched_param *restrict param);
int pthread_setschedprio(pthread_t thread, int prio);

```

The pthread\_setschedparam() function sets both the scheduling policy and priority of a thread directly without the use of an attribute object. The thread parameter is the id of the thread, policy is the new scheduling policy, and param contains the scheduling priority. The pthread\_getschedparam() function shall return the scheduling policy and scheduling parameters and store their values in policy and param parameters, respectively, if successful. If successful, both functions will return 0. If not successful, both functions will return an error number. [Table 4-7](#) lists the conditions in which these functions may fail.

The pthread\_setschedprio() function is used to set the scheduling priority of an executing thread whose thread id is specified by the thread parameter. The scheduling priority of the thread will be changed to the value specified by prio. If the function fails, the priority of the thread will not be changed. If successful, the function will return 0. If not successful, an error number is returned. The conditions in which this function fails are also listed in [Table 4-7](#).

**Table 4-7. Conditions in Which the Scheduling Policy and Priority Functions May Fail**

**Pthread Scheduling and Priority Failure Conditions Functions**

- |                           |  |
|---------------------------|--|
| int pthread_getschedparam | <ul style="list-style-type: none"> <li>• The thread parameter does not refer to an existing</li> </ul> |
|---------------------------|--|

## Pthread Scheduling and Priority Failure Conditions Functions

```
(pthread_t thread, thread.  
int *restrict policy,  
struct sched_param  
*restrict param) ;
```

```
int pthread_setschedparam  
(pthread_t thread,  
int *policy,  
const struct sched_param  
*parm);
```

- The policy parameter or one of the scheduling parameters associated with the policy parameter is invalid.
- The policy parameter or one of the scheduling parameters has a value that is not supported.
- The calling thread does not have the appropriate permission to set the scheduling parameters or policy of the specified thread.
- The thread parameter does not refer to an existing thread.
- The implementation does not allow the application to change one of the parameters to the specified value.

```
int pthread_setschedprio  
(pthread_t thread,  
int prio) ;
```

- The prio parameter is invalid for the scheduling policy of the specified thread.
- The priority parameter has a value that is not supported.
- The calling thread does not have the appropriate permission to set the scheduling priority of the specified thread.
- The thread parameter does not refer to an existing thread.
- The implementation does not allow the application to change the priority to the specified value.

### NOTE:

Remember to carefully consider why it is necessary to change the scheduling policy or priority of a running thread. This may diversely affect the overall performance of your application. Threads with higher priority preempt running threads with lower priority. This may lead to starvation, or a thread constantly being preempted and therefore not able to complete execution.

#### 4.9.3.1 Setting Contention Scope of a Thread

The contention scope of the thread determines which set of threads with the same scheduling policy and priority, the thread will compete for processor usage. The contention scope of a thread is set by the thread attribute object.

### Synopsis

```
#include <pthread.h>

int pthread_attr_setscope(pthread_attr_t *attr, int contentscope);
int pthread_attr_getscope(const pthread_attr_t *restrict attr,
                          int *restrict contentscope);
```

The `pthread_attr_setscope()` function sets the contention scope attribute of the thread attribute object specified by the parameter `attr`. The contention scope of the thread attribute object will be set to the value stored in the `contentscope` parameter. The `contentscope` parameter can have the values:

<code>PTHREAD_SCOPE_SYSTEM</code>	System scheduling contention scope
<code>PTHREAD_SCOPE_PROCESS</code>	Process scheduling contention scope

The `pthread_attr_getscope()` function returns the contention scope attribute from the thread attribute object specified by the parameter `attr`. If successful, the contention scope of the thread attribute object will be returned and stored in the `contentscope` parameter. Both functions return 0 if successful and an error number otherwise.

#### 4.9.4 Using `sysconf()`

It is important to know the thread resource limits of your system in order for your application to appropriately manage its resources. For example, the maximum number of threads per process places an upper bound on the number of worker threads that can be created for a process. The `sysconf()` function is used to return the current value of configurable system limits or options.

### Synopsis

```
#include <unistd.h>
#include <limits.h>

int sysconf(int name);
```

The `name` parameter is the system variable to be queried. What is returned is the POSIX IEEE Std. 1003.1-2001 values for the system variable queried. These values can be compared to the constants defined by your implementation of the standard to see how compliant they are. There are several variables and constant counterparts concerned with threads, processes, and semaphores, some of which are listed in [Table 4-8](#).

The `sysconf()` function will return -1 and set `errno` to indicate an error has occurred if the parameter `name` is not valid. The variable may have no limit defined and may return -1 as a valid return value. In that case, `errno` will not be set. No defined limit does not mean there is an infinite limit. It simply indicates that no maximum limit is defined and higher limits are supported depending upon the system

resources available.

Here is an example of a call to the `sysconf()` function:

```
if(PTHREAD_STACK_MIN == (sysconf(_SC_THREAD_STACK_MIN))){  
    //...  
}
```

The constant value of `PTHREAD_STACK_MIN` is compared to the `_SC_THREAD_STACK_MIN` value returned by the `sysconf()` function.

**Table 4-8. Systems Variables and Their Corresponding Symbolic Constants**

<b>Variable</b>	<b>Name Value</b>	<b>Description</b>
<code>_SC_THREADS</code>	<code>_POSIX_THREADS</code>	Supports threads.
<code>_SC_THREAD_ATTR_STACKADDR</code>	<code>_POSIX_THREAD_ATTR_STACKADDR</code>	Supports thread stack address attribute.
<code>_SC_THREAD_ATTR_STACKSIZE</code>	<code>_POSIX_THREAD_ATTR_STACKSIZE</code>	Supports thread stack size attribute.
<code>_SC_THREAD_STACK_MIN</code>	<code>PTHREAD_STACK_MIN</code>	Minimum size of thread stack storage in bytes.
<code>_SC_THREAD_THREADS_MAX</code>	<code>PTHREAD_THREADS_MAX</code>	Maximum number of threads per process.
<code>_SC_THREAD_KEYS_MAX</code>	<code>PTHREAD_KEYS_MAX</code>	Maximum number of keys per process.
<code>_SC_THREAD_PRIORITY_INHERIT</code>	<code>_POSIX_THREAD_PRIORITY_INHERIT</code>	Supports priority inheritance option.
<code>_SC_THREAD_PRIORITY</code>	<code>_POSIX_THREAD_PRIORITY</code>	Supports thread priority option.
<code>_SC_THREAD_PRIORITY_SCHEDULING</code>	<code>_POSIX_THREAD_PRIORITY_SCHEDULING</code>	Supports thread priority scheduling option.
<code>_SC_THREAD_PROCESS_SHARED</code>	<code>_POSIX_THREAD_PROCESS_SHARED</code>	Supports process-shared synchronization.

<b>Variable</b>	<b>Name Value</b>	<b>Description</b>
<code>_SC_THREAD_SAFE_FUNCTIONS</code>	<code>_POSIX_THREAD_SAFE_FUNCTIONS</code>	Supports thread-safe functions.
<code>_SC_THREAD_DESTRUCTOR_ITERATIONS</code>	<code>_PTHREAD_THREAD_DESTRUCTOR_ITERATIONS</code>	Determines the number of attempts made to destroy thread-specific data on thread exit.
<code>_SC_CHILD_MAX</code>	<code>CHILD_MAX</code>	Maximum number of processes allowed to a UID.
<code>_SC_PRIORITY_SCHEDULING</code>	<code>_POSIX_PRIORITY_SCHEDULING</code>	Supports process scheduling.
<code>_SC_REALTIME_SIGNALS</code>	<code>_POSIX_REALTIME_SIGNALS</code>	Supports real-time signals.
<code>_SC_XOPEN_REALTIME_THREADS</code>	<code>_XOPEN_REALTIME_THREADS</code>	Supports X/Open POSIX real-time threads feature group.
<code>_SC_STREAM_MAX</code>	<code>STREAM_MAX</code>	Determines the number of streams one process can have open at a time.
<code>_SC_SEMAPHORES</code>	<code>_POSIX_SEMAPHORES</code>	Supports semaphores.
<code>_SC_SEM_NSEMS_MAX</code>	<code>SEM_NSEMS_MAX</code>	Determines the maximum number of semaphores a process may have.
<code>_SC_SEM_VALUE_MAX</code>	<code>SEM_VALUE_MAX</code>	Determines the maximum value a semaphore may have.
<code>_SC_SHARED_MEMORY_OBJECTS</code>	<code>_POSIX_SHARED_MEMORY_OBJECTS</code>	Supports shared memory objects.

#### 4.9.5 Managing a Critical Section

Concurrently executing processes, or threads within the same process, can share data structures, variables, or data. Sharing global memory allows the processes or threads to communicate or share access to data. With multiple processes, the shared global memory is external to the processes that the processes in question have access. This data structure can be used to transfer data or commands among the processes. When threads need to communicate, they can access data structures or variables that are part of the same process to which they belong.

Whether there are processes or threads accessing shared modifiable data, the data structures, variables, or data is in a critical region or section of the processes' or threads' code. A critical section in the code is where the thread or process is accessing and processing the shared block of modifiable memory. Classifying a section of code as a critical section can be used to control race conditions. For example, in a program two threads, thread A and thread B, are used to perform a multiple keyword search through all the files located on a system. Thread A searches each directory for text files and writes the paths to a list data structure TextFiles then increments a FileCount variable. Thread B extracts the filenames from the list TextFiles, decrements the FileCount, then searches the file for the multiple keywords. The file that contains the keywords is written to a file and another variable, FoundCount, is incremented. FoundCount is not shared with thread A. Threads A and B can be executed simultaneously on separate processors. Thread A executes until all directories have been searched while thread B searches each file extracted from TextFiles. The list is maintained in sorted order and can be requested to display its contents any time.

A number of problems can crop up. For example, thread B may attempt to extract a filename from TextFiles before thread A has added a filename to TextFiles. Thread B may attempt to decrement SearchCount before thread A has incremented SearchCount or both may attempt to modify the variable simultaneously. Also TextFiles may be sorting its elements while thread A is simultaneously attempting to write a filename to it or thread B is simultaneously attempting to extract a filename from it. These problems are examples of race conditions in which two or more threads or processes are attempting to modify the same block of shared memory simultaneously.

When threads or processes are simply simultaneously reading the same block of memory, race conditions do not occur. Race conditions occur when multiple processes or threads are simultaneously accessing the same block of memory with at least one of the threads or processes attempting to modify the block of memory. The section of code becomes critical when there are simultaneous attempts to change the same block of memory. One way to protect the critical section is to only allow exclusive access to the block of memory. Exclusive access means one process or thread will have access to the shared block of memory for a short period while all other processes or threads are prevented (blocked) from entering their critical section where they are accessing the same block of memory.

A locking mechanism, like a mutex semaphore, can be used to control race condition. A mutex, short for "mutual exclusion," is used to block off a critical section. The mutex is locked before entering the critical section then unlocked when exiting the critical section:

```
lock mutex
    // enter critical section
    // access shared modifiable memory
    // exit critical section
unlock mutex
```

The pthread\_mutex\_t models a mutex object. Before the pthread\_mutex\_t object can be used, it must first be initialized. The pthread\_mutex\_init() initializes the mutex. Once initialized the mutex can be locked, unlocked, and destroyed with the pthread\_mutex\_lock(), pthread\_mutex\_unlock(), and pthread\_mutex\_destroy() functions. [Program 4.5](#) contains the function that searches a system for text files. [Program 4.6](#) contains the function that searches each text file for specified keywords. Each function is executed by a thread. [Program 4.7](#) contains the primary thread. These programs implement the producer-consumer model for thread delegation. [Program 4.5](#) contains the producer thread and [Program 4.6](#) contains the consumer thread. The critical sections are bolded.

#### Program 4.5

```
1 int isDirectory(string FileName)
2 {
```



```

3  struct stat StatBuffer;
4
5  lstat(FileName.c_str(),&StatBuffer);
6  if((StatBuffer.st_mode & S_IFDIR) == -1)
7  {
8      cout << "could not get stats on file" << endl;
9      return(0);
10 }
11 else{
12     if(StatBuffer.st_mode & S_IFDIR){
13         return(1);
14     }
15 }
16 return(0);
17 }
18
19
20 int isRegular(string FileName)
21 {
22     struct stat StatBuffer;
23
24     lstat(FileName.c_str(),&StatBuffer);
25     if((StatBuffer.st_mode & S_IFDIR) == -1)
26     {
27         cout << "could not get stats on file" << endl;
28         return(0);
29     }
30     else{
31         if(StatBuffer.st_mode & S_IFREG){
32             return(1);
33         }
34     }
35     return(0);
36 }
37
38
39 void depthFirstTraversal(const char *CurrentDir)
40 {
41     DIR *DirP;
42     string Temp;
43     string FileName;
44     struct dirent *EntryP;
45     chdir(CurrentDir);
46     cout << "Searching Directory: " << CurrentDir << endl;
47     DirP = opendir(CurrentDir);
48
49     if(DirP == NULL){
50         cout << "could not open file" << endl;
51         return;
52     }
53     EntryP = readdir(DirP);
54     while(EntryP != NULL)
55     {
56         Temp.erase();
57         FileName.erase();
58         Temp = EntryP->d_name;
59         if((Temp != ".") && (Temp != "..")){
60             FileName.assign(CurrentDir);
61             FileName.append(1, '/');
62             FileName.append(EntryP->d_name);

```

```

63     if(isDirectory(FileName)){
64         string NewDirectory;
65         NewDirectory = FileName;
66         depthFirstTraversal(NewDirectory.c_str());
67     }
68     else{
69         if(isRegular(FileName)){
70             int Flag;
71             Flag = FileName.find(".cpp");
72             if(Flag > 0){
73                 pthread_mutex_lock(&CountMutex);
74                 FileCount++;
75                 pthread_mutex_unlock(&CountMutex);
76                 pthread_mutex_lock(&QueueMutex);
77                 TextFiles.push(FileName);
78                 pthread_mutex_unlock(&QueueMutex);
79             }
80         }
81     }
82 }
83 }
84 EntryP = readdir(DirP);
85 }
86 closedir(DirP);
87 }
88
89
90
91 void *task(void *X)
92 {
93     char *Directory;
94     Directory = static_cast<char *>(X);
95     depthFirstTraversal(Directory);
96     return(NULL);
97 }
98 }

```

[Program 4.6](#) contains the consumer thread that performs the search.

#### Program 4.6

```

1 void *keySearch(void *X)
2 {
3     string Temp, Filename;
4     less<string> Comp;
5
6     while(!Keyfile.eof() && Keyfile.good())
7     {
8         Keyfile >> Temp;
9         if(!Keyfile.eof()){
10            KeyWords.insert(Temp);
11        }
12    }
13    Keyfile.close();
14
15    while(TextFiles.empty())
16    { }
17
18    while(!TextFiles.empty())

```

```

19 {
20     pthread_mutex_lock(&QueueMutex);
21     Filename = TextFiles.front();
22     TextFiles.pop();
23     pthread_mutex_unlock(&QueueMutex);
24     Infile.open(Filename.c_str());
25     SearchWords.erase(SearchWords.begin(),SearchWords.end());
26
27     while(!Infile.eof() && Infile.good())
28     {
29         Infile >> Temp;
30         SearchWords.insert(Temp);
31     }
32
33     Infile.close();
34     if(includes(SearchWords.begin(),SearchWords.end(),
35                Keywords.begin(),Keywords.end(),Comp)){
36         Outfile << Filename << endl;
37         pthread_mutex_lock(&CountMutex);
38         FileCount--;
39         pthread_mutex_unlock(&CountMutex);
40         FoundCount++;
41     }
42     return(NULL);
43 }
44 }

```

[Program 4.7](#) contains the primary thread for producer–consumer threads in [Programs 4.5](#) and [4.6](#).

#### Program 4.7

```

1 #include <sys/stat.h>
2 #include <fstream>
3 #include <queue>
4 #include <algorithm>
5 #include <pthread.h>
6 #include <iostream>
7 #include <set>
8
9 pthread_mutex_t QueueMutex = PTHREAD_MUTEX_INITIALIZER;
10 pthread_mutex_t CountMutex = PTHREAD_MUTEX_INITIALIZER;
11
12 int FileCount = 0;
13 int FoundCount = 0;
14
15 int keySearch(void);
16 queue<string> TextFiles;
17 set <string,less<string> >Keywords;
18 set <string,less<string> >SearchWords;
19 ifstream Infile;
20 ofstream Outfile;
21 ifstream Keyfile;
22 string KeywordFile;
23 string OutFilename;
24 pthread_t Thread1;
25 pthread_t Thread2;
26
27 void depthFirstTraversal(const char *CurrentDir);

```

```

28 int isDirectory(string FileName);
29 int isRegular(string FileName);
30
31 int main(int argc, char *argv[])
32 {
33     if(argc != 4){
34         cerr << "need more info" << endl;
35         exit (1);
36     }
37
38     Outfile.open(argv[3],ios::app||ios::ate);
39     Keyfile.open(argv[2]);
40     pthread_create(&Thread1,NULL,task,argv[1]);
41     pthread_create(&Thread2,NULL,keySearch,argv[1]);
42     pthread_join(Thread1,NULL);
43     pthread_join(Thread2,NULL);
44     pthread_mutex_destroy(&CountMutex);
45     pthread_mutex_destroy(&QueueMutex);
46
47     cout << argv[1] << " contains " << FoundCount
48         << " files that contains all keywords." << endl;
49 }

```

With mutexes, one thread at a time is permitted to read from or write to the shared memory. There are other mechanisms and techniques that can be used to ensure thread safety for user-defined functions implementing one of the PRAM models:

- EREW (exclusive read and exclusive write)
- CREW (concurrent read and exclusive write)
- ERCW (exclusive read and concurrent write)
- CRCW (concurrent read and concurrent write)

Mutexes are used to implement EREW algorithms, which will be discussed in [Chapter 5](#).

## 4.10 Thread Safety and Libraries

According to Klieman, Shah, and Smaalders (1996): "A function or set of functions is said to be thread safe or reentrant when the functions may be called by more than one thread at a time without requiring any other action on the caller's part." When designing a multithread application, the programmer must be careful to ensure that concurrently executing functions are thread safe. We have already discussed making user-defined functions thread safe but an application often calls functions defined by the system- or a third-party-supplied library. Some of these functions and/or libraries are thread safe where others are not. If the functions are not thread safe, then this means the functions contain one or more of the following: static variables, accesses global data, and/or is not reentrant.

If the function contains static variables, then those variables maintain their values between invocations of the function. The function requires the value of the static variable in order to operate correctly. When concurrent multiple threads invoke this function, then a race condition occurs. If the function modifies a global variable, then multiple threads invoking that function may each attempt to modify that global variable. If multiple concurrent accesses to the global variable are not synchronized, then a race condition can occur here as well. For example, multiple concurrent threads can execute functions that set `errno`. With some of the threads, the function fails and `errno` is set to an error message while other threads execute successfully. Depending on the compiler implementation, `errno` is thread safe. If not,

when a thread checks the state of `errno`, which message will it report?

A block of code is considered reentrant if the code cannot be changed while in use. Reentrant code avoids race conditions by removing references to global variables and modifiable static data. Therefore, the code can be shared by multiple concurrent threads or processes without a race condition occurring. The POSIX standard defines several functions as reentrant. They are easily identified by a `_r` attached to the function name of the nonreentrant counterpart. Some are listed below:

```
getgrgid_r()
getgrnam_r()
getpwuid_r()
sterror_r()
strtok_r()
readdir_r()
rand_r()
ttyname_r()
```

If the function accesses unprotected global variables; contains static, modifiable variables; or is not reentrant, then the function is considered thread unsafe.

System- or third-party-supplied libraries may have different versions of their standard libraries. One version is for single-threaded applications and the other version for multithreaded applications. Whenever a multithreaded environment is anticipated, the programmer should link to these multithreaded versions of the library. Other environments do not require multithreaded applications to be linked to the multithreaded version of the library but only require macros to be defined in order for reentrant versions of functions to be declared. The application will then be compiled as thread safe.

It is not possible in all cases to use multithreaded versions of functions. In some instances, multithreaded versions of particular functions are not available for a given compiler or environment. Some function's interface cannot be simply made thread safe. In addition, the programmer may be faced with adding threads to an environment that uses functions that were only meant to be used in a single-threaded environment. Under these conditions, in general use mutexes to wrap all such functions within the program. For example, a program has three concurrently executing threads. Two of the threads, `thread1` and `thread2`, both concurrently execute `funcA()`, which is not thread safe. The third thread, `thread3`, executes `funcB()`. To solve the problem of `funcA()`, the solution may be to simply wrap access to `funcA()` by `thread1` and `thread2` with a mutex:

```
thread1      thread2      thread3
{            {            {
  lock()      lock()          funcB()
  funcA()     funcA()
  unlock()    unlock()
}            }            }
```

If this is done then only one thread accesses `funcA()` at a time. But there is still a problem. If `funcA()` and `funcB()` are both thread-unsafe functions, they may both modify a global or static variable. Although `thread1` and `thread2` are using mutexes with `funcA()`, `thread3` will be executing `funcB()` concurrently with either of these threads. In this situation, a race condition occurs because `funcA()` and `funcB()` may both modify the same global or static variable.

To illustrate another type of race condition when dealing with the `iostream` library, let's say we have two threads, `thread A` and `thread B`, sending output to the standard output stream, `cout`. `cout` is an object of type `ostream`. Using inserters, (`>>`), and extractors, (`<<`), invokes the methods of the `cout` object. Are these methods thread safe? If `thread A` is sending the message "We are intelligent beings" to `stdout` and `thread B` is sending the message "Humans are illogical beings," will the output be interleaved and produce a message "We are Humans are illogical beings intelligent beings"? In some cases, thread-safe

functions are implemented as atomic functions. Atomic functions are functions that once they begin to execute cannot be interrupted. In the case of cout, if the inserter operation is implemented as atomic, then this interweaving cannot take place. When there are multiple calls to the inserter operation, they will be executed as if they were in serial order. Thread A's message will be displayed, then thread B's, or vice versa, although they invoked the function simultaneously. This is an example of serializing a function or operation in order to make it thread safe. This may not be the only way to make a function thread safe. A function may interweave operations if it has no adverse effect. For example, if a method adds or removes elements to or from a structure that is not sorted and two different threads invoke that method, interweaving their operations will not have an adverse effect.

If it is not known which functions from a library are thread safe and which are not, the programmer has three choices:

- Restrict use of all thread-unsafe functions to a single thread.
- Do not use any of the thread-unsafe functions.
- Wrap all potential thread-unsafe functions within a single set of synchronization mechanisms.

An additional approach is to create interface classes for all thread-unsafe functions that will be used in a multithreaded application. The unsafe functions are encapsulated within an interface class. The interface class can be combined with the appropriate synchronization objects through inheritance or composition. The interface class can be used by the host class through inheritance or composition. The approach eliminates the possibility of race conditions.

## 4.11 Dividing Your Program into Multiple Threads

Earlier in this chapter we discussed the delegation of work according to a specific strategy or approach called a thread model. Those thread models were:

- delegation (boss–worker)
- peer-to-peer
- pipeline
- producer–consumer

Each model has its own WBS (Work Breakdown Structure) that determines who is responsible for thread creation and under what conditions threads are created. In this section we will show an example of a program for each model using Pthread library functions.

### 4.11.1 Using the Delegation Model

We discussed two approaches that can be used to implement the delegation approach to dividing a program into threads. To recall, in the delegation model, a single thread (boss) creates the threads (workers) and assigns each a task. The boss thread delegates the task each worker thread is to perform by specifying a function. With one approach, the boss thread creates threads as a result of requests made to the system. The boss thread processes each type of request in an event loop. As events occur, thread workers are created and assigned their duties. [Example 4.5](#) shows the event loop in the boss thread and the worker threads in pseudocode.

**Example 4.5 Approach 1: Skeleton program of boss and worker thread model.**

```
//...
pthread_mutex_t Mutex = PTHREAD_MUTEX_INITIALIZER
int AvailableThreads
pthread_t Thread[Max_Threads]
void decrementThreadAvailability(void)
void incrementThreadAvailability(void)
int threadAvailability(void);

// boss thread
{
    //...
    if(sysconf(_SC_THREAD_THREADS_MAX) > 0){
        AvailableThreads = sysconf(_SC_THREAD_THREADS_MAX)
    }
    else{
        AvailableThreads = Default
    }

    int Count = 1;

    loop while(Request Queue is not empty)
        if(threadAvailability()){
            Count++
            decrementThreadAvailability()
            classify request
            switch(request type)
            {
                case X : pthread_create(&(Thread[Count])...taskX...)
                case Y : pthread_create(&(Thread[Count])...taskY...)
                case Z : pthread_create(&(Thread[Count])...taskZ...)
                //...
            }
        }
        else{
            //free up thread resources
        }
    end loop
}

void *taskX(void *X)
{
    // process X type request
    incrementThreadAvailability()
    return(NULL)
}

void *taskY(void *Y)
{
    // process Y type request
    incrementThreadAvailability()
    return(NULL)
}

void *taskZ(void *Z)
{
    // process Z type request
```

```

    decrementThreadAvailability()
    return(NULL)
}

//...

```

In [Example 4.5](#), the boss thread dynamically creates a thread to process each new request that enters the system, but there are a maximum number of threads that will be created. There are n number of tasks to process n request types. To be sure the maximum number of threads per process will not be exceeded, these additional functions can be defined:

```

threadAvailability()
incrementThreadAvailability()
decrementThreadAvailability()

```

[Example 4.6](#) shows pseudocode for these functions.

**Example 4.6 Functions that manage thread availability count.**

```

void incrementThreadAvailability(void)
{
    //...
    pthread_mutex_lock(&Mutex)
    AvailableThreads++
    pthread_mutex_unlock(&Mutex)
}

void decrementThreadAvailability(void)
{
    //...
    pthread_mutex_lock(&Mutex)
    AvailableThreads--
    pthread_mutex_unlock(&Mutex)
}

int threadAvailability(void)
{
    //...
    pthread_mutex_lock(&Mutex)
    if(AvailableThreads > 1)
        return 1
    else
        return 0
    pthread_mutex_unlock(&Mutex)
}

```

The threadAvailability() function will return 1 if the maximum number of threads allowed per process has not been reached. This function accesses a global variable ThreadAvailability that stores the number of threads still available for the process. The boss thread calls the decrementThreadAvailability() function, which decrements the global variable before the boss thread creates a thread. The worker threads call incrementThreadAvailability(), which increments the global variable before a worker thread exits. Both functions contain a call to pthread\_mutex\_lock() before accessing the variable and a call to pthread\_mutex\_unlock() after accessing the global variable. If the maximum number of threads are exceeded, then the boss thread can cancel threads if possible or spawn another process, if necessary. taskX(), taskY(), and taskZ() execute code that processes their type of request.



The other approach to the delegation model is to have the boss thread create a pool of threads that are reassigned new requests instead of creating a new thread per request. The boss thread creates a number of threads during initialization and then each thread is suspended until a request is added to the queue. The boss thread will still contain an event loop to extract requests from the queue. But instead of creating a new thread per request, the boss thread signals the appropriate thread to process the request. [Example 4.7](#) shows the boss thread and the worker threads in pseudocode for this approach to the delegation model.

**Example 4.7 Approach 2: Skeleton program of boss and worker thread model.**

```
//...

pthread_t Thread[N]

// boss thread
{
    pthread_create(&(Thread[1]...taskX...));
    pthread_create(&(Thread[2]...taskY...));
    pthread_create(&(Thread[3]...taskZ...));
    //...

    loop while(Request Queue is not empty
        get request
        classify request
        switch(request type)
        {
            case X :
                enqueue request to XQueue
                signal Thread[1]

            case Y :
                enqueue request to YQueue
                signal Thread[2]

            case Z :
                enqueue request to ZQueue
                signal Thread[3]

            //...
        }
    end loop
}

void *taskX(void *X)
{
    loop
        suspend until awoken by boss
        loop while XQueue is not empty
            dequeue request
            process request

    end loop
until done
}
```

```

void *taskY(void *Y)
{
    loop
        suspend until awoken by boss
        loop while YQueue is not empty
            dequeue request
            process request
        end loop
    until done
}

void *taskZ(void *Z)
{
    loop
        suspend until awoken by boss
        loop while (ZQueue is not empty)
            dequeue request
            process request
        end loop
    until done
}

//...

```

In [Example 4.7](#), the boss thread creates N number of threads, one thread for each task to be executed. Each task is associated with processing a request type. In the event loop, the boss thread dequeues a request from the request queue, determines the request type, enqueues the request to the appropriate request queue, then signals the thread that processes the request in that queue. The functions also contain an event loop. The thread is suspended until it receives a signal from the boss that there is a request in its queue. Once awakened, in the inner loop, the thread processes all the requests in the queue until it is empty.

#### 4.11.2 Using the Peer-to-Peer Model

In the peer-to-peer model, a single thread initially creates all the threads needed to perform all the tasks called peers. The peer threads process requests from their own input stream. [Example 4.8](#) shows a skeleton program of the peer-to-peer approach of dividing a program into threads.

##### Example 4.8 Skeleton program using the peer-to-peer model

```

//...

pthread_t Thread[N]

// initial thread
{
    pthread_create(&(Thread[1]...taskX...));
    pthread_create(&(Thread[2]...taskY...));
    pthread_create(&(Thread[3]...taskZ...));
    //...
}

```

```

void *taskX(void *X)
{
    loop while (Type XRequests are available)
        extract Request
        process request
    end loop
    return(NULL)
}

//...

```

In the peer-to-peer model, each thread is responsible for its own stream of input. The input can be extracted from a database, file, and so on.

### 4.11.3 Using the Pipeline Model

In the pipeline model, there is a stream of input processed in stages. At each stage, work is performed on a unit of input by a thread. The input continues to move to each stage until the input has completed processing. This approach allows multiple inputs to be processed simultaneously. Each thread is responsible for producing its interim results or output, making them available to the next stage or next thread in the pipeline. [Example 4.9](#) shows the skeleton program for the pipeline model.

**Example 4.9** Skeleton program using the pipeline model.

```

//...

pthread_t Thread[N]
Queues[N]

// initial thread
{
    place all input into stage1's queue
    pthread_create(&(Thread[1]...stage1...));
    pthread_create(&(Thread[2]...stage2...));
    pthread_create(&(Thread[3]...stage3...));
    //...
}

void *stageX(void *X)
{
    loop
        suspend until input unit is in queue
        loop while XQueue is not empty
            dequeue input unit
            process input unit
            enqueue input unit into next stage's queue
        end loop
    until done
    return(NULL)
}

//...

```

In [Example 4.9](#), N queues are declared for N stages. The initial thread enqueues all the input into stage 1's queue. The initial thread then creates all the threads needed to execute each stage. Each stage has an event loop. The thread sleeps until an input unit has been enqueued. The inner loop continues to iterate until its queue is empty. The input unit is dequeued, processed, then that unit is then enqueued into the

queue of the next stage.

#### 4.11.4 Using the Producer–Consumer Model

In the producer-consumer model, the producer thread produces data consumed by the consumer thread or threads. The data is stored in a block memory shared between the producer and consumer threads. This model was used in [Programs 4.5](#), [4.6](#), and [4.7](#). [Example 4.10](#) shows the skeleton program for the producer-consumer model.

##### Example 4.10 Skeleton program using the producer–consumer model.

```
pthread_mutex_t Mutex = PTHREAD_MUTEX_INITIALIZER
pthread_t Thread[2]
Queue

// initial thread
{
    pthread_create(&(Thread[1]...producer...));
    pthread_create(&(Thread[2]...consumer...));
    //...
}

void *producer(void *X)
{
    loop
        perform work
        pthread_mutex_lock(&Mutex)
        enqueue data
        pthread_mutex_unlock(&Mutex)
        signal consumer
        //...
    until done
}

void *consumer(void *X)
{
    loop
        suspend until signaled
        loop while(Data Queue not empty)
            pthread_mutex_lock(&Mutex)
            dequeue data
            pthread_mutex_unlock(&Mutex)
            perform work
        end loop
    until done
}
```

In [Example 4.9](#), an initial thread creates the producer and consumer threads. The producer thread executes a loop in which it performs work then locks a mutex on the shared queue in order to enqueue the data it has produced. The producer unlocks the mutex then signals the consumer thread that there is data in the queue. The producer iterates through the loop until all work is done. The consumer thread also executes a loop in which it suspends itself until it is signaled. In the inner loop, the consumer thread processes all the data until the queue is empty. It locks the mutex on the shared queue before it dequeues any data and unlocks the mutex after the data has been dequeued. It then performs work on that data. In [Program 4.6](#), the consumer thread enqueues its results to a file. The results could have been inserted into another data structure. This is often done by consumer threads in which it plays both the

role of consumer and producer. It plays the role of consumer of the unprocessed data produced by the producer thread, then it plays the role of producer when it processes data stored in another shared queue consumed by another thread.

#### 4.11.5 Creating Multithreaded Objects

The delegation, peer-to-peer, pipeline, and producer–consumer models demonstrate approaches to dividing a program into multiple threads along function lines. When using objects, member functions can create threads to perform multiple tasks. Threads can be used to execute code on behalf of the object: free-floating functions and other member functions.

In either case, the threads are declared within the object and created by one of the member functions (e.g., the constructor). The threads can then execute some free-floating functions (function defined outside the object), which invokes member functions of the object that are global. This is one approach to making an object multithreaded. [Example 4.10](#) contains an example of a multithreaded object.

##### Example 4.10 Declaration and definition of multithreading an object.

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>

void *task1(void *);
void *task2(void *);

class multithreaded_object
{
    pthread_t Thread1,Thread2;
public:

    multithreaded_object(void);
    int c1(void);
    int c2(void);
    //...
};

multithreaded_object::multithreaded_object(void)
{
    //...
    pthread_create(&Thread1,NULL,task1,NULL);
    pthread_create(&Thread2,NULL,task2,NULL);
    pthread_join(Thread1,NULL);
    pthread_join(Thread2,NULL);
    //...
}

int multithreaded_object::c1(void)
{
    // do work
    return(1);
}
```

```

int multithreaded_object::c2(void)
{
    // do work
    return(1);
}

multithreaded_object MObj;

void *task1(void *)
{
    //...
    MObj.c1();
    return(NULL);
}

void *task2(void *)
{
    //...
    MObj.c2();
    return(NULL);
}

```

In [Example 4.10](#), the class `multithread_object` declares two threads. From the constructor of the class, the threads are created and joined. Thread1 executes `task1` and Thread2 executes `task2`. `task1` and `task2`, then invokes member functions of the global object `MObj`.

## Summary

In a sequential program, work can be divided between routines within a program where one task finishes then another task can perform work. With other programs, work is executed as mini-programs within the main program where the mini-programs execute concurrently with the main program. These mini-programs can be executed as processes or threads. With processes, each process has its own address space and requires interprocess communication if the processes are to communicate. Threads sharing the address space of the process do not require special communication techniques between threads of the same process. Synchronization mechanisms such as mutexes are needed to protect share memory in order to control race conditions.

There are several models that can be used to delegate work among threads and manage when threads are created and canceled. In the delegation model, a single thread (boss) creates the threads (workers) and assigns each a task. The boss thread waits until each worker thread completes its task. With the peer-to-peer model, there is a single thread that initially creates all the threads needed to perform all the tasks; that thread is considered a worker thread and does no delegation. All threads have equal status. The pipeline model is characterized as an assembly line in which a stream of items are processed in stages. At each stage, a thread executes work performed on the unit of input. The input moves from one thread to the next, processing it until completion. The last stage or thread produces the result of the pipeline. In the producer–consumer model, there is a producer thread that produces data to be consumed by the consumer thread. The data is stored in a block of memory shared between the producer and consumer threads. Objects can be made to be multithreaded. The threads are declared within the object. A member function can create a thread that executes a free-floating function that in turn invokes one of the member functions of the object.

The Pthread library can be used to create and manage the threads of a multithreaded application. The Pthread library is based on a standardized programming interface for the creation and maintenance of threads. The thread interface has been specified by the IEEE standards committee in the POSIX 1003.1c standard. Third-party vendors supply an implementation that adheres to the POSIX standard.

## Chapter 5. Synchronizing Concurrency between Tasks

"The relation of these mechanisms to time demands careful study. ... We are scarcely ever interested in the performance of a communication-engineering machine for a single input. To function adequately, it must give a satisfactory performance for a whole class of inputs, and this means a statistically satisfactory performance for the class of input which it is statistically expected to receive . . ."

—Norbert Wiener, Cybernetics

In this Chapter

- [Coordinating Order of Execution](#)
- [Synchronizing Access to Data](#)
- [What are Semaphores?](#)
- [Synchronization: An Object-Oriented Approach](#)
- [Summary](#)

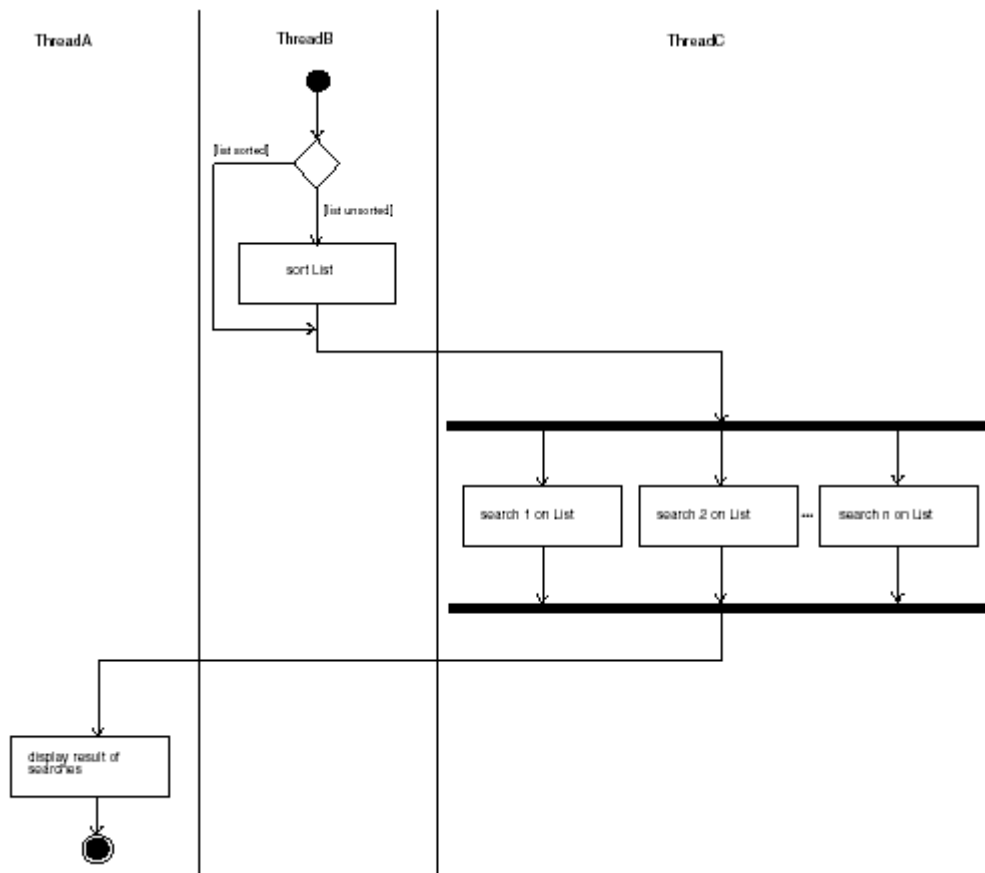
With any computer system, resources are limited. There is only so much memory, I/O devices and ports, hardware interrupts, and processors. In an environment of limited hardware resources, an application consisting of multiple processes and threads must compete for memory locations, peripheral devices, and processor time. It is the job of the operating system to determine when the process or thread utilizes system resources and for how long. With preemptive scheduling, the operating system can interrupt the process or thread in order to accommodate all the processes and threads competing for the system resources. Processes and threads must also compete for software and data resources. An example of software resources is shared libraries that provide a common set of services or functions to processes and threads. Other shareable software resources are applications, programs, and utilities. When sharing software resources, only one copy of the program(s) code is brought into memory. Data resources are objects, system data (e.g., environment variables) files, globally defined variables, and data structures. With data resources, it is possible for processes and threads to have their own copy. In other cases, it is desirable and maybe necessary that data is shared. Some processes and threads work together to use the system's limited resources while other processes and threads work independently and asynchronously, competing for the use of the shareable resource. There are several techniques and mechanisms that can be used by the programmer to manage competing processes and threads to share data resources.

Synchronization is also needed to coordinate the order of execution of concurrent tasks. The producer-consumer model discussed in [Chapter 4](#) is a prime example. It is necessary for the producer to execute before the consumer, not necessarily finish before the consumer. Synchronization is required to coordinate these tasks in order for work to progress. Data (access synchronization) and task synchronization (sequence synchronization) are two types of synchronization required when executing multiple concurrent tasks.

## 5.1 Coordinating Order of Execution

Let's say we have three threads executing concurrently labeled thread A, thread B, and thread C. All three threads are involved in list processing. The list is to be sorted and searched and the results displayed. Each thread is assigned a task; thread A is to display the results of the search, thread B is to sort the list, and thread C is to search the list. First, the list has to be sorted then multiple concurrent searches can occur on the list. The results of the searches are then displayed. If these threads' tasks are not synchronized properly, thread A may attempt to display results not yet generated that violates the postcondition of the process. The precondition in the list must be sorted prior to searching. If searches start before the list is sorted, the search may generate the wrong results. The three threads require task synchronization. Task synchronization enforces preconditions and postconditions of logical processes. [Figure 5-1](#) shows a UML activity diagram for this process.

**Figure 5-1. Activity diagram for sorting, searching, and displaying the contents of a list.**



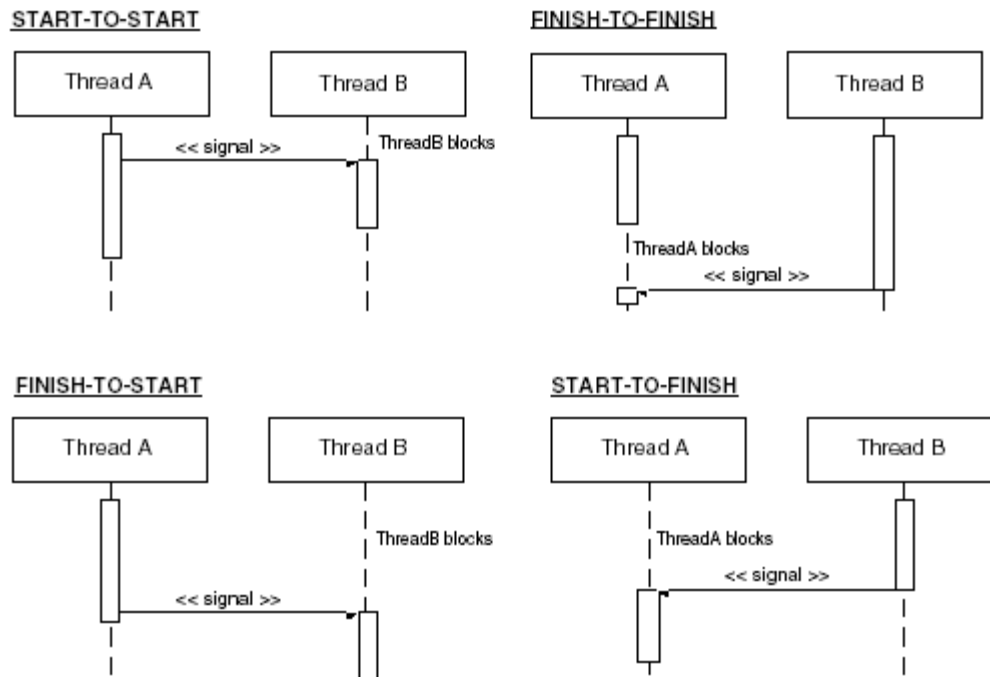
The thread B's sort must occur first, then forking to the multiple searches spawned by thread C takes place. The threads are then joined and thread A displays the results.

### 5.1.1 Relationships between Synchronized Tasks

There are four basic synchronization relationships between any two threads in a single process or between any two processes within a single application: start-to-start (SS), finish-to-start (FS), start-to-finish (SF), and finish-to-finish (FF). These four basic relationships characterize the coordination of work between threads and processes. [Figure 5-2](#) shows activity diagrams for each synchronization relationship.



Figure 5-2. The synchronization relationships that can exist between tasks A and B.



### 5.1.2 Start-to-Start (SS) Relationship

In a start-to-start synchronization relationship, one task cannot start until another task starts. One task may start before the other but never after. For example, let's say we have a program that implements an avatar. The avatar is a computer-generated talking head. The avatar provides a kind of personality for the software. The program that implements the avatar has several threads. Here, we will focus on thread A, which controls the animation of the mouth and thread B, which controls the sound or voice. We want to give the illusion that the sound and mouth animation are synchronized. Ideally, they should execute at the same precise moment. If multiple processors are involved, both threads may start simultaneously. The threads have a start-to-start relationship. Because of timing conditions, it is allowed that thread A start slightly before thread B (not much before for illusion's sake) but thread B cannot start before thread A. The voice has to wait for the animation. It is not desirable to hear a voice before the mouth animates (unless it is simulating voice dubbing).

### 5.1.3 Finish-to-Start (FS) Relationship

In a finish-to-start synchronization relationship, task A cannot finish until task B starts. This type of relationship is common with parent-child processes. The parent process cannot complete execution of some operation until it spawns a child process or it receives a communication from the child process that it has started its operation. The child process continues to execute once it has signaled the parent or supplied the needed information. The parent process is then free to complete its operation.

### 5.1.4 Start-to-Finish Relationship

A start-to-finish synchronization relationship is the reverse of the finish-to-start relationship. In a start-to-finish synchronization relationship, one task cannot start until another task finishes. Task A cannot start execution until task B finishes executing or completes a certain operation. If process A is reading from a pipe connected to process B, process B must first write to the pipe before process A reads from it. Process B must at least complete one operation, writing a single element to the pipe before process A starts. The producer-consumer threads in [Chapter 4](#) are another example of a finish-to-start relationship.

The sort-search threads in [Figure 5-1](#) also demonstrate this relationship. The sort thread had to complete its work before the search threads were to search the list. In all these cases, one thread or process has to complete an operation before another thread or process attempts to execute its operation. Unless this work is coordinated, the goal of the process, thread, or application would fail or give inaccurate results.

The finish-to-start relationship usually suggests there is an information dependency between the tasks. With information dependency, interthread or interprocess communication is required from one or more tasks in order for a thread or process to operate correctly. The search would produce incorrect results unless the sort was performed. The consumer thread would have no files to process unless the producer thread produced the files to be searched.

### 5.1.5 Finish-to-Finish Relationship

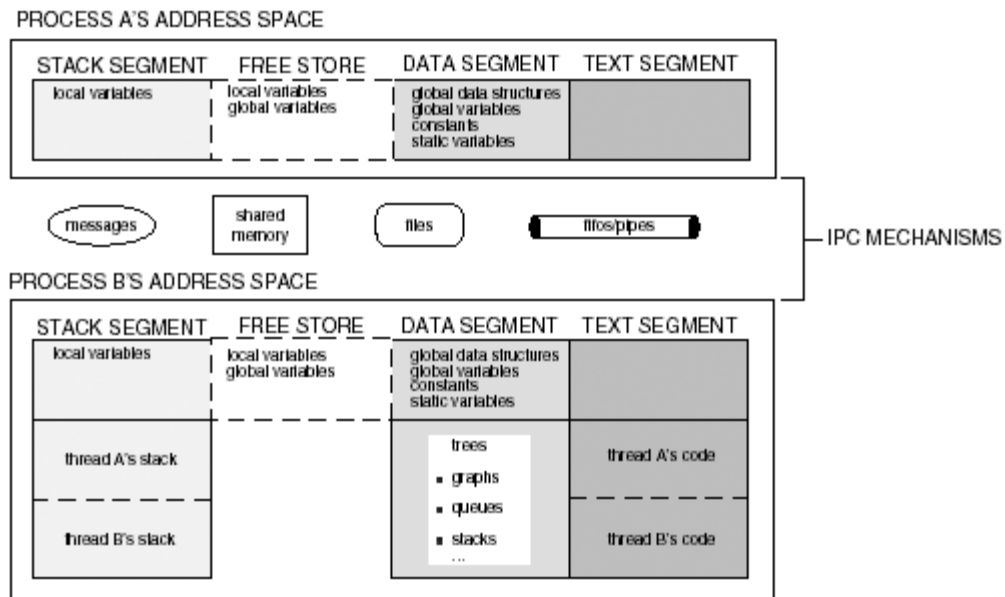
In a finish-to-finish synchronization relationship, one task cannot finish until another task finishes. Task A cannot finish until task B finishes. This again can describe the relationship between parent and child processes discussed in [Chapter 3](#). The parent process must wait until all its child processes have terminated before it is allowed to terminate. If the parent process terminates before its child processes, those terminated child processes become zombied. Parent processes should not finish (exit the system in this case) until all its child processes have finished executing. The parent process achieves this by either calling a `wait()` function for each of its child processes, or waiting for a mutex or condition variable that can be broadcast by child threads. Another example of a finish-to-finish relationship is the boss-worker model. The boss thread's job is to delegate work to the worker threads. It would be undesirable for the boss thread to terminate before the worker threads. New requests to the system would not be processed, existing threads would have no work to perform, and no new threads would be created. If the boss thread is the primary thread and it terminates, the process would terminate along with all the worker threads. In a peer-to-peer model, if thread A dynamically allocates an object passed to thread B and thread A terminates, the object is destroyed along with thread A. If this is done before thread B has had a chance to use it, a segmentation fault or data access violation will occur. In order to prevent these kinds of errors with threads, termination of threads is synchronized by using the `pthread_join()` function. A call to this function causes the calling thread to wait on the target thread until it finishes. This creates finish-to-finish synchronization.

## 5.2 Synchronizing Access to Data

There is a difference between data shared between processes and data shared between threads. Threads share the same address space. Processes have separate address spaces. If there are two processes, A and B, then data declared in process A is not available to process B and vice versa. Therefore, one method used by processes to share data is to create a block of memory that is then mapped to the address space of the processes that are to share the memory. Another approach is to create a block of memory that exists outside the address space of both processes. These are types of IPC (interprocess communication) that include: pipes, files, and message passing.

It is the block of memory shared between threads within the same address space and the block of memory shared between processes outside both address spaces that requires data synchronization. [Figure 5-3](#) contrasts memory shared between threads and processes.

**Figure 5-3. The memory shared between threads and processes.**



Data synchronization is needed in order to control race conditions and allow concurrent threads or processes to safely access a block of memory. Data synchronization controls when a block of memory can be read or modified. Concurrent access to shared memory, global variables, and files must be synchronized in a multithreaded environment. Data synchronization is needed at the location in a task's code when it attempts to access the block of memory, global variable, or file shared with other concurrently executing processes or threads. This block of code is called the critical section. The critical section can be any block of code that changes the file pointer's position, writes to the file, closes the file, and reads or writes global variables or data structures. Classifying the tasks as read or write tasks is one step in managing concurrent access to the shared memory.

### 5.2.1 PRAM Model

The PRAM (Parallel Random-Access Machine) is a simplified theoretical model where there are  $N$  processors, labeled as  $P_1, P_2, P_3, \dots, P_n$ , share one global memory. All the processors have simultaneous read and write access to shared global memory. Each of these theoretical processors can access the global shared memory in one uninterruptible unit of time. The PRAM model has concurrent read and write algorithms and exclusive read and write algorithms. Concurrent read algorithms are allowed to read the same piece of memory simultaneously with no data corruption. Concurrent write algorithms allow multiple processors to write to the shared memory. Exclusive read algorithms are used to ensure that no two processors ever read the same memory location at the same time. Exclusive write ensures that no two processors write to the same memory at the same time. The PRAM model can be used to characterize concurrent access to shared memory by multiple tasks.

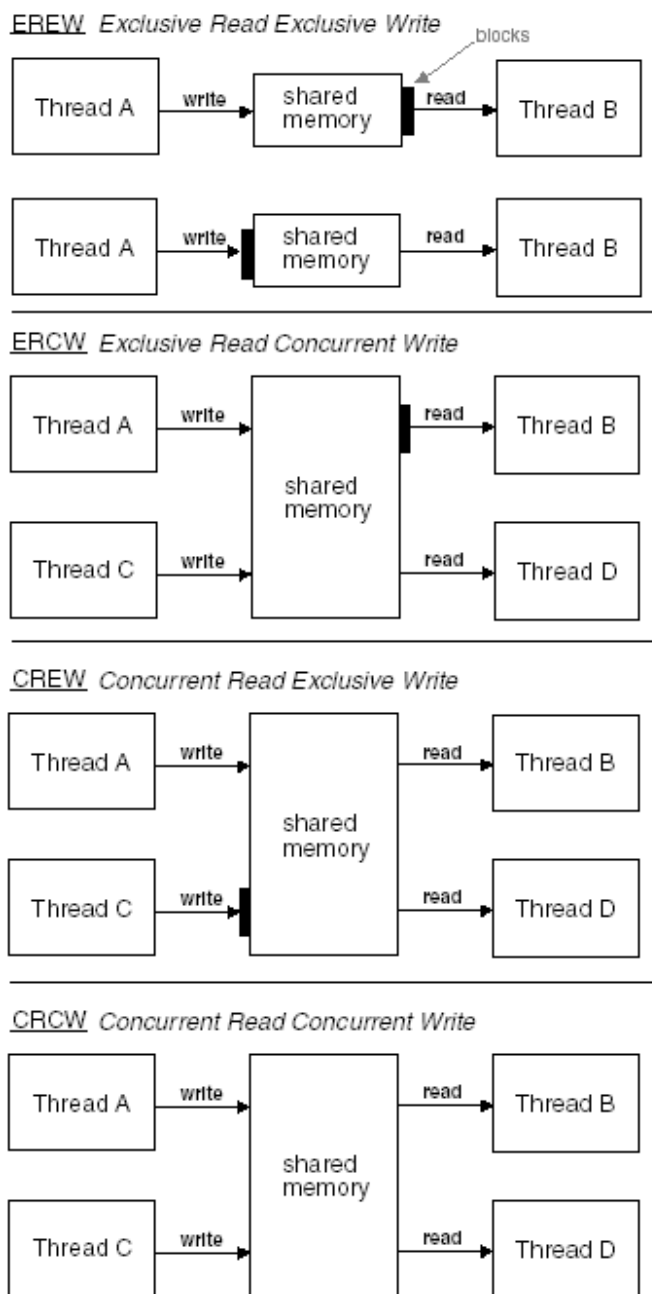
#### 5.2.1.1 Concurrent and Exclusive Memory Access

The concurrent and exclusive read-write algorithms can be combined into the following types of algorithm combinations that are possible for read-write access:

- EREW (exclusive read and exclusive write)
- CREW (concurrent read and exclusive write)
- ERCW (exclusive read and concurrent write)
- CRCW (concurrent read and concurrent write)

These algorithms can be viewed as the access policy implemented by the tasks sharing the data. [Figure 5-4](#) illustrates these access policies. EREW means access to the shared memory is serialized. Only one task at a time is given access to the shared memory. An example of EREW access policy is the producer-consumer example discussed in [Chapter 4](#). Access to the queue that contained the filenames was restricted to exclusive write by the producer and exclusive read by the consumer. Only one task was allowed access to the queue at any given time. CREW access policy allows multiple reads of the shared memory and exclusive writes. This means there are no restrictions on how many tasks can read the shared memory concurrently but only one task can write to the shared memory. Concurrent reads can occur while an exclusive write is taking place. With this type of access policy, each reading task may read a different value. As a task reads the shared memory, another task modifies it. The next task that reads the shared memory will see different data. The ERCW access policy is the direct reverse of CREW. With ERCW, concurrent writes are allowed but only one task at a time is allowed to read the shared memory. CRCW access policy allows concurrent reads and concurrent writes.

**Figure 5-4. EREW, CREW, ERCW, and CRCW access policies.**



Each of these four algorithm types requires different levels and types of synchronization. They can be analyzed on a continuum with the access policy that requires the least amount of synchronization to implement on one end and the access policy that requires the most amount of synchronization at the other end. The goal is to implement these policies and maintain data integrity and satisfactory system performance. EREW is the policy that is the simplest to implement. This is because EREW essentially forces sequential processing. At first blush, you may consider CRCW is the simplest but it presents the most challenges. It may appear as if it has no policy. The memory can be accessed without restriction. But this is the furthest from the truth. This is the most difficult to implement and requires the most synchronization in order to meet our goal.

## 5.3 What are Semaphores?

A semaphore is a synchronization mechanism that can be used to manage synchronization relationships and implement the access policies. A semaphore is a special kind of variable that can only be accessed by very specific operations. The semaphore is used to help threads and processes synchronize access to shared modifiable memory or manage access to a device or other resource. The semaphore is used as a key to access the resource. This key can only be owned by one process or thread at a time. Whichever task owns the key or semaphore locks the resource for its exclusive use. Locking the resource causes any other task that wishes to use the resource to wait until the resource has been unlocked, making it available again. Once unlocked, the next task waiting for the semaphore is given the semaphore, thus accessing the resource. The next task is determined by the scheduling policy in effect for that thread or process.

### 5.3.1 Semaphore Operations

As mentioned earlier, a semaphore can only be accessed by specific operations like an object. There are two operations that can be performed on a semaphore. The P() operation is a decrement operation and the V() operation is an increment operation. If Mutex is the semaphore, then here are the logical implementations of the P(Mutex) and V(Mutex) operations:

P(Mutex)

```
if(Mutex > 0) {
    Mutex--;
}
else {
    Block on Mutex;
}
```

V(Mutex)

```
if(Blocked on Mutex N processes) {
    pass on Mutex;
}
else{
    Mutex++;
}
```

The actual implementation will be system dependent. These operations are indivisible, meaning once the operation is in progress, it cannot be preempted. If several tasks attempt to make a call to the P() operation, only one task will be allowed to proceed. If the Mutex has already been decremented, then the task will block and be placed in a queue. The V() operation is called by the task that has the Mutex. If other tasks are waiting on the Mutex, it is given to the next task in the queue. If no tasks are waiting, then the Mutex is incremented.

Semaphore operations can go by other names:

P() operation:

V() operation:

lock()

unlock()

The value of the semaphore will depend on the type of semaphore it is. There are several types of semaphores. A binary semaphore will have the value 0 or 1. A counting semaphore will have some non-negative integer value.

The POSIX standard defines several types of semaphores. These semaphores are used by processes or threads. [Table 5-1](#) lists the types of semaphores. The table also lists some of their basic operations.

**Table 5-1. Semaphore Types Defined by the POSIX Standard and Their Use by Threads and/or Processes**

<b>Types of Semaphores</b>	<b>Processes/Threads</b>	<b>Description</b>
Mutex semaphores	Processes or threads	Mechanism used to implement mutual exclusion in a critical section of code.
Read–write locks	Processes or threads	Mechanism used to implement read-write access policy between threads.
Condition variables	Processes or threads	Mechanism used to broadcast a signal between threads that an event has taken place.  When a thread locks an event mutex, it blocks until it receives the broadcast.
Multiple condition variables	Processes or threads	Same as an event mutex but includes multiple events or conditions.

Any operating system that is compliant with the Single UNIX Specification or POSIX Standard will supply an implementation of these semaphores. They are a part of the libpthread library and the functions are declared in the pthread.h header.

### 5.3.2 Mutex Semaphores

The POSIX standard defines a mutex semaphore used by threads and processes of type `pthread_mutex_t`. This mutex provides the basic operations necessary to make it a practical synchronization mechanism:

- initialization
- request ownership
- release ownership

- try ownership
- destruction

[Table 5-2](#) lists the `pthread_mutex_t` functions that are used to perform these basic operations. The initialization process allocates memory required to hold the mutex semaphore and give the memory some initial values. For a binary semaphore, its initial value will be 0 or 1. If it's a counting semaphore, its initial value is a non-negative number that represents the number of resources available. It can be used to represent the request limit a program is capable of processing in a single session. Unlike regular variables, there is no guarantee that the initialization operation of a mutex will occur. After calling the initialization operation, take precautions to ensure that the mutex was initialized (i.e., checking the return value or checking the `errno` value). The system shall fail to create the mutex if the space set aside for mutexes has been used, the number of allowable semaphores will be exceeded, the named semaphore already exists, or there is some other memory allocation problem.

**Table 5-2. pthread\_mutex\_t Functions**

Mutex Operations	Function Prototypes/Macros #include <pthread.h>
Initialization	<pre>int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;</pre>
Request ownership	<pre>&lt;time.h&gt;  int pthread_mutex_lock(pthread_mutex_t *mutex);  int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const struct timespec *restrict abs_timeout);</pre>
Release ownership	<pre>int pthread_mutex_unlock(pthread_mutex_t *mutex);</pre>
Try ownership	<pre>int pthread_mutex_trylock(pthread_mutex_t *mutex);</pre>
Destruction	<pre>int pthread_mutex_destroy(pthread_mutex_t *mutex);</pre>

Similar to a thread, the Pthread mutex has an attribute object that encapsulates all the attributes of the mutex. This mutex attribute will be discussed later. It can be passed to the initialization function, creating a mutex with attributes of those set in the mutex object. If no attribute object is used, the mutex will be initialized with default values. The `pthread_mutex_t` is initialized as unlocked and private. A private mutex is shared between threads of the same process. A shared mutex is shared between threads of multiple processes. If default attributes are to be used, the mutex can be initialized statically for statically allocated mutex objects by using the macro:

```
pthread_mutex_t Mutex = PTHREAD_MUTEX_INITIALIZER;
```

This method uses less overhead but performs no error checking.

A mutex can be owned or unowned. The request ownership operation grants ownership of the mutex to

the calling process or thread. Once the mutex is owned, the thread or process has exclusive access to the resource. If there is any attempt to own the mutex (by calling this operation) by any other processes or threads, they are blocked until the mutex is made available. Releasing the mutex causes the next process or thread that has blocked on this mutex to unblock and obtain ownership of the mutex. With `pthread_mutex_lock()`, the thread granted ownership of a given mutex is the only thread that can release the mutex. A timed version of this function is also available. In that case, if the mutex is owned the process or thread will wait for some specified period of time. If the mutex is not released in that time interval, the process or thread will continue executing.

The try ownership operation tests the mutex to see if it is already owned. If owned, the function returns some value indicating that. The advantage of this operation is the thread or process is not blocked if the mutex is owned. It will be able to continue executing. If the mutex is not owned, then ownership is granted.

The destruction operation frees the memory associated with the mutex. The memory cannot be destroyed or closed if it is owned or a thread or process is waiting for the mutex.

### 5.3.2.1 Using the Mutex Attribute Object

The `pthread_mutex_t` has an attribute object used in a similar way as the thread attribute. The attribute object encapsulates all the attributes of a mutex object. Once initialized, it can be used by multiple mutex objects when passed to the `pthread_mutex_init()` function. The mutex attribute defines several functions used to set these attributes: priority ceiling, protocol, and type. These functions and other attribute functions are listed in [Table 5-3](#) with a brief description.

**Table 5-3. pthread\_mutex\_t Attribute Object Functions**

<b>pthread_mutex_t Attribute Object Function Prototypes #include &lt;pthread.h&gt;</b>	<b>Description</b>
<pre>int pthread_mutexattr_init (pthread_mutexattr_t * attr);</pre>	Initializes a mutex attribute object specified by the parameter <code>attr</code> with default values for all of the attributes defined by the implementation.
<pre>int pthread_mutexattr_destroy (pthread_mutexattr_t * attr);</pre>	Destroys a mutex attribute object specified by the parameter <code>attr</code> , which causes the mutex attribute object to become uninitialized. Can be reinitialized by calling the <code>pthread_mutexattr_init()</code> function.
<pre>int pthread_mutexattr_ setprioceiling (pthread_mutexattr_t * attr,  int prioceiling);</pre>	Sets and returns the priority ceiling attribute of the mutex specified by the parameter <code>attr</code> . The parameter <code>prioceiling</code> contains the priority ceiling of the mutex. The <code>prioceiling</code> attribute defines the minimum priority level at which the critical section guarded by the mutex is executed. The values are within the maximum range of priorities defined by <code>SCHED_FIFO</code> .
<pre>int pthread_mutexattr_ getprioceiling (const pthread_mutexattr_t *  restrict attr, int *restrict  prioceiling);</pre>	



**pthread\_mutex\_t Attribute Object  
Function Prototypes #include  
<pthread.h>**

**Description**

```
int pthread_mutexattr_  
setprotocol  
(pthread_mutexattr_t * attr,  
 int protocol);
```

Sets and returns the protocol of the mutex attribute specified by the parameter attr. The protocol parameter contains the value of the protocol attribute:

```
int pthread_mutexattr_  
getprotocol  
(const pthread_mutexattr_t *  
 restrict attr,  
 int *restrict protocol);
```

PTHREAD\_PRIO\_NONE

The priority and scheduling of the thread is not affected by the ownership of the mutex.

PTHREAD\_PRIO\_INHERIT

Thread blocking other threads of higher priority due to ownership of such a mutex, shall execute at the highest priority of any of the threads waiting on any of the mutexes owned by this thread with such a protocol.

PTHREAD\_PRIO\_PROTECT

Threads owning such a mutex shall execute at the highest priority ceilings of all mutexes owned by this thread with such a protocol, regardless of whether other threads are blocked on any of these mutexes.

```
int pthread_mutexattr_  
setpshared  
(pthread_mutexattr_t * attr,  
 int pshared);
```

Sets or returns the process-shared attribute of the mutex attribute object specified by the parameter attr. The pshared parameter contains a value:

```
int pthread_mutexattr_  
getpshared  
(const pthread_mutexattr_t *  
 restrict attr, int *restrict  
 pshared);
```

PTHREAD\_PROCESS\_SHARED

Permits a mutex to be shared by any threads that have access to the allocated memory of the mutex even if the threads are in different processes.

PTHREAD\_PROCESS\_PRIVATE

Mutex is shared between threads of the same process as the initialized mutex.

```
int pthread_mutexattr_  
settype  
(pthread_mutexattr_t * attr,
```

Sets and returns the type mutex attribute of the mutex attribute specified by the parameter attr. The mutex type

**pthread\_mutex\_t Attribute Object**  
**Function Prototypes #include**  
**<pthread.h>**

**Description**

```
int type);
```

attribute is used to describe the behavior of the mutex, which includes whether the mutex will determine deadlock, perform error checking, etc. The type parameter contains a value:

```
int pthread_mutexattr_  
gettype  
(const pthread_mutexattr_t *  
restrict attr,  
int *restrict type);
```

PTHREAD\_MUTEX\_DEFAULT

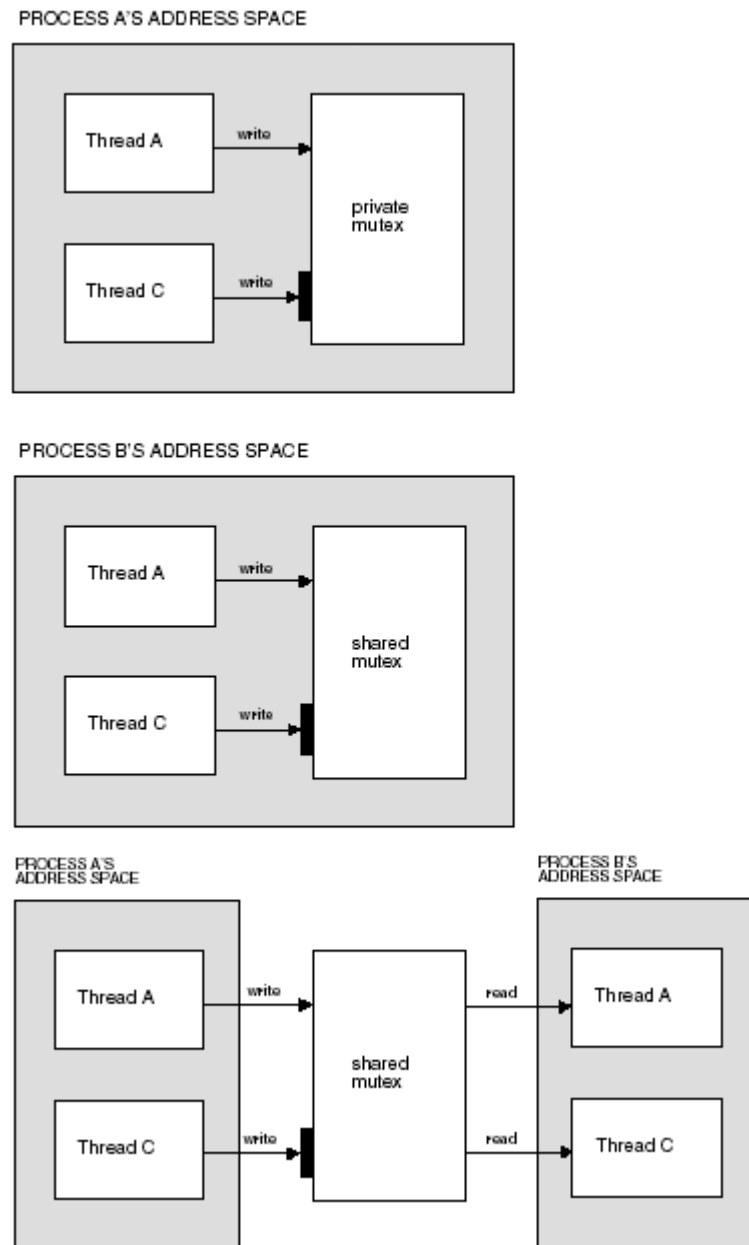
PTHREAD\_MUTEX\_RECURSIVE

PTHREAD\_MUTEX\_ERRORCHECK

PTHREAD\_MUTEX\_NORMAL

The most interesting of the attributes is setting whether the mutex will be private or shared. Private mutexes are only shared among threads of the same process. It can be declared as global or a handle can be passed between threads. Shared mutexes are used by any threads that have access to the memory in which the mutex is located. This includes threads of different processes. [Figure 5-5](#) contrasts the idea of private and shared mutexes between different processes. If threads of different processes are to share a mutex, it must be allocated in memory shared between processes. POSIX defines several functions used to allocate shared memory between objects using memory-mapped files and shared memory objects. Mutexes between processes can be used to protect critical sections that access files, pipes, shared memory, and devices.

Figure 5-5. Private and shared mutexes.



### 5.3.2.2 Using Mutex Semaphores to Manage Critical Sections

Mutexes can be used to manage critical sections of processes and threads in order to control race conditions. Mutexes avoid race conditions by serializing access to the critical section. [Example 5.1](#) shows two threads. Mutexes are used to protect their critical sections.

**Example 5.1 Using mutexes to protect critical sections of threads.**

```
// ...
pthread_t ThreadA,ThreadB;
pthread_mutex_t Mutex;
pthread_mutexattr_t MutexAttr;

void *task1(void *X)
{
    pthread_mutex_lock(&Mutex);
```

```

    // critical section of code
    pthread_mutex_unlock(&Mutex);
    return(0);
}

void *task2(void *X)
{
    pthread_mutex_lock(&Mutex);
    // critical section of code
    pthread_mutex_unlock(&Mutex);
    return(0);
}

int main(void)
{
    //...
    pthread_mutexattr_init(&MutexAttr);
    pthread_mutex_init(&Mutex,&MutexAttr);
    //set mutex attributes
    pthread_create(&ThreadA,NULL,task1,NULL);
    pthread_create(&ThreadB,NULL,task2,NULL);
    //...
    return(0);
}

```

In [Example 5.1](#), ThreadA and ThreadB have critical sections protected by their use of Mutex.

[Example 5.2](#) shows how mutexes can be used to protect the critical sections of currently executing processes.

**Example 5.2 Mutexes used to protect critical sections.**

```

//...
int Rt;
pthread_mutex_t Mutex1;
pthread_mutexattr_t MutexAttr;

int main(void)
{
    //...
    pthread_mutexattr_init(&MutexAttr);
    pthread_mutexattr_setpshared(&MutexAttr,
                                PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(&Mutex1,&MutexAttr);

    if((Rt = fork()) == 0){ // child process
        pthread_mutex_lock(&Mutex1);
        //critical section
        pthread_mutex_unlock(&Mutex1);
    }
    else{ // parent process
        pthread_mutex_lock(&Mutex1);
        //critical section
        pthread_mutex_unlock(&Mutex1);
    }
    //...
    return(0);
}

```

```
}
```

In [Example 5.2](#), it is important to note that the mutex has been initialized as shared by calling:

```
pthread_mutexattr_setshared(&MutexAttr, PTHREAD_PROCESS_SHARED);
```

This allows `Mutex` to be shared by threads of different processes. Once `fork()` is called, the child process and parent process can protect their critical section with `Mutex`. Their critical sections will contain some resource shared by both processes.

### 5.3.3 Read–Write Locks

Mutex semaphores are used to manage a critical section by serializing entry to that section. Only one thread or process is permitted to enter the critical section at a time. With read-write locks, multiple threads are allowed to enter the critical section if they are to read the shared memory only. Therefore, any number of threads can own a read-write lock for reading. But if multiple threads are to write or modify the shared memory, only one thread is given access. No other threads are allowed to enter the critical section if one thread is given exclusive access to write to the shared memory. This can be used when applications more often read data than write data. If the application has multiple threads, mutex exclusion can be extreme. The performance of the application can benefit by allowing multiple reads. The POSIX standard defines a read-write lock of type `pthread_rwlock_t`.

Similar to mutex semaphores, the read-write locks have the same operations. [Table 5-4](#) lists the read-write lock operations.

The difference between regular mutexes and read-write mutexes is their locking request operations. Instead of one locking operation there are two:

```
pthread_rwlock_rdlock()  
pthread_rwlock_wrlock()
```

`pthread_rwlock_rdlock()` obtains a read-lock and `pthread_rwlock_wrlock()` obtains a write lock. If a thread requests a read lock, it is granted the lock as long as there are no threads that hold a write lock. If so, the calling thread is blocked. If a thread request a write lock, it is granted as long as there are no threads that hold a read lock or a write lock. If so, the calling thread is blocked.

The read-write lock is of type `pthread_rwlock_t`. This type also has an attribute object that encapsulates its attributes. The attribute functions are listed in [Table 5-5](#).

**Table 5-4. Read–Write Lock Operations**

<b>Read–write Lock Operations</b>	<b>Function Prototypes #include &lt;pthread.h&gt;</b>
-----------------------------------	---

Initialization	<pre>int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);</pre>
----------------	---

Request ownership	<pre>#include &lt;time.h&gt;  int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock); int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock); int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock, const struct timespec *restrict abs_timeout);</pre>
-------------------	---

## Read–write Lock Operations    **Function Prototypes #include <pthread.h>**

```
int pthread_rwlock_timedwrlock(pthread_rwlock_t |
*restrict rwlock, const struct timespec *restrict
abs_timeout);
```

Release ownership                    `int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);`

Try ownership                        `int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);`

```
int pthread_rwlock_trywrlock(pthread_rwlock_t
*rwlock);
```

Destruction                         `int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);`

The `pthread_rwlock_t` can be private between threads or shared between threads or different processes.

### 5.3.3.1 Using Read-Write Locks to Implement Access Policy

Read-write locks can be used to implement an access policy, namely CREW. Several tasks can be granted concurrent reads but only one task is granted write access. Using read-write locks will not permit concurrent reads to occur with the exclusive write. [Example 5.3](#) contains tasks using read-write locks to protect critical sections.

**Table 5-5. Attribute Object Functions for `pthread_rwlock_t`**

#### **pthread\_rwlock\_t Attribute Object Function Description Prototypes #include <pthread.h>**

<code>int pthread_rwlockattr_init(pthread_rwlockattr_t * attr);</code>	Initializes a read-write lock attribute object specified by the parameter <code>attr</code> with default values for all of the attributes defined by the implementation.
<code>int pthread_rwlockattr_destroy(pthread_rwlockattr_t * attr);</code>	Destroys a read-write lock attribute object specified by the parameter <code>attr</code> . Can be reinitialized by calling the <code>pthread_rwlockattr_init()</code> function.
<code>int pthread_rwlockattr_setpshared(pthread_rwlockattr_t * attr, int pshared);</code>	Sets or returns the process-shared attribute of the read-write lock attribute object specified by the parameter <code>attr</code> . The <code>pshared</code> parameter contains a value:
<code>int pthread_rwlockattr_getpshared</code>	<code>PTHREAD_PROCESS_SHARED</code>

## pthread\_rwlock\_t Attribute Object Function Description

Prototypes #include <pthread.h>

```
(const pthread_rwlockattr_t *  
 restrict attr,  
 int *restrict pshared);
```

Permits a read-write lock to be shared by any threads that have access to the allocated memory of the read-write lock even if the threads are in different processes.

PTHREAD\_PROCESS\_PRIVATE

The read-write lock is shared between threads of the same process as the initialized rwlock.

### Example 5.3 Threads using read-write locks.

```
//...  
pthread_t ThreadA,ThreadB,ThreadC,ThreadD;  
pthread_rwlock_t RWLock;
```

```
void *producer1(void *X)  
{  
    pthread_rwlock_wrlock(&RWLock);  
    //critical section  
    pthread_rwlock_unlock(&RWLock);  
    return(0);  
}
```

```
void *producer2(void *X)  
{  
    pthread_rwlock_wrlock(&RWLock);  
    //critical section  
    pthread_rwlock_unlock(&RWLock);  
}
```

```
void *consumer1(void *X)  
{  
    pthread_rwlock_rdlock(&RWLock);  
    //critical section  
    pthread_rwlock_unlock(&RWLock);  
    return(0);  
}
```

```
void *consumer2(void *X)  
{  
    pthread_rwlock_rdlock(&RWLock);  
    //critical section  
    pthread_rwlock_unlock(&RWLock);  
    return(0);  
}
```

```

int main(void)
{
    pthread_rwlock_init(&RWLock, NULL);
    //set mutex attributes
    pthread_create(&ThreadA, NULL, producer1, NULL);
    pthread_create(&ThreadB, NULL, consumer1, NULL);
    pthread_create(&ThreadC, NULL, producer2, NULL);
    pthread_create(&ThreadD, NULL, consumer2, NULL);
    //...
    return(0);
}

```

In [Example 5.3](#), four threads are created. Two threads are producers, ThreadA and ThreadC, and two threads are consumers, ThreadB and ThreadD. All the threads have a critical section and each section is protected with the read–write lock RWLock. As mentioned, ThreadB and ThreadD can enter their critical sections concurrently or serially but neither thread can enter their critical sections if either ThreadA or ThreadC is in theirs. ThreadA and ThreadC cannot enter their critical sections concurrently. [Table 5-6](#) shows part of the decision table for [Example 5.3](#).

**Table 5-6. Part of the Decision Table for [Example 5.3](#)**

Thread A (writer)	Thread B (reader)	Thread C (writer)	Thread D (reader)
N	N	N	Y
N	N	Y	N
N	Y	N	N
N	Y	N	Y
Y	N	N	N

### 5.3.4 Condition Variables

A condition variable is a semaphore used to signal an event has occurred. One or more processes or threads can wait for the signal sent by other processes or threads once the event has taken place. Some make a distinction between condition variables and the mutex semaphores discussed. The purpose of the mutex semaphore and read–write locks is to synchronize data access whereas condition variables are typically used to synchronize the sequence of operations. W. Richard Stevens, in his book UNIX Network Programming, states it best: "Mutexes are for locking and cannot be used for waiting."

In [Program 4.6](#), our consumer thread contained a busy loop:

```

15 while(TextFiles.empty())
16 {}

```

The consumer thread looped until there were items in the TextFiles queue. This can be replaced by a condition variable. The producer can signal the consumer that items have been inserted into the queue.



The consumer can wait until it receives the signal then continue to process the queue.

The condition variable is of type `pthread_cond_t`. These are the types of operations it can perform:

- initialize
- destroy
- wait
- timed wait
- signal
- broadcast

The initialize and destroy operations work in a similar manner as the other mutexes. [Table 5-7](#) lists the functions for the `pthread_cond_t` that implement these operations.

**Table 5-7. Functions for the `pthread_cond_t` that Implement Condition Variables Operations**

**Condition Variables Operations    Function Prototypes/Macros #include <pthread.h>**

Initialization	<pre>int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;</pre>
----------------	--

Waiting	<pre>int pthread_cond_wait(pthread_cond_t * restrict cond, pthread_mutex_t *restrict mutex);  int pthread_cond_timedwait(pthread_cond_t * restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);</pre>
---------	---

Signaling	<pre>int pthread_cond_signal(pthread_cond_t *cond); int pthread_cond_broadcast(pthread_cond_t *cond);</pre>
-----------	---

Destruction	<pre>int pthread_cond_destroy(pthread_cond_t *cond);</pre>
-------------	--

Condition variables are used in conjunction with mutexes. If a thread or process attempts to lock a mutex, we know that it will block until the mutex is released. Once unblocked, it obtains the mutex then continues. If a condition variable is used, it must be associated with a mutex.

```
//...
pthread_mutex_lock(&Mutex);
pthread_cond_wait(&EventMutex,&Mutex);
//...
pthread_mutex_unlock(&Mutex);
```

A task attempts to lock a mutex. If the mutex is already locked then the task will block. Once unblocked, the task will release the mutex, `Mutex`, while it waits on the signal for the condition variable, `EventMutex`. If the mutex is not locked, it will release the mutex and wait indefinitely. With a timed wait, the task will only wait for a specified period of time. If the time expires before the task is signaled, the function will return an error. It will then reacquire the mutex.

The signal operation causes a task to signal to another thread or process that an event has occurred. If a task is waiting for that condition variable, it will be unblocked and given the mutex. If there are several tasks waiting for the condition variable, only one will be unblocked. The tasks will be waiting in a queue and unblocked according to the scheduling policy. The broadcast operation signals all the task waiting for the condition variable. If multiple tasks are unblocked, the tasks shall compete for the ownership of the mutex according to a scheduling policy. In contrast to the wait operation, the signaling task is not required to own the mutex, although it is recommended.

The condition variable also has an attribute object. [Table 5-8](#) lists the functions of the attribute object with a brief description.

#### 5.3.4.1 Using Condition Variables to Manage Synchronization Relationships

The condition variable can be used to implement the synchronization relationships mentioned earlier: start-to-start (SS), finish-to-start (FS), start-to-finish (SF), and finish-to-finish (FF). These relationships can exist between threads of the same processes or different processes. [Examples 5.4](#) and [5.5](#) contain examples of how to implement an FS and FF synchronization relationship. There are two mutexes used in each example. One mutex is used to synchronize access to the shared data and the other mutex is used to synchronize execution of code.

##### Example 5.4 FS synchronization relationship between two threads.

```
//...
float Number;
pthread_t ThreadA,ThreadB;
pthread_mutex_t Mutex,EventMutex;
pthread_cond_t Event;

void *worker1(void *X)
{
    for(int Count = 1;Count < 100;Count++){
        pthread_mutex_lock(&Mutex);
        Number++;
        pthread_mutex_unlock(&Mutex);
        cout << "worker1: number is " << Number << endl;
        if(Number == 50){
            pthread_cond_signal(&Event);
        }
    }
    cout << "worker 1 done" << endl;
    return(0);
}

void *worker2(void *X)
{
    pthread_mutex_lock(&EventMutex);
    pthread_cond_wait(&Event,&EventMutex);
    pthread_mutex_unlock(&EventMutex);
    for(int Count = 1;Count < 50;Count++){
        pthread_mutex_lock(&Mutex);
        Number = Number + 20;
        pthread_mutex_unlock(&Mutex);
        cout << "worker2: number is " << Number << endl;
    }
    cout << "worker 2 done" << endl;
    return(0);
}
```

```

int main(int argc, char *argv[])
{
    pthread_mutex_init(&Mutex, NULL);
    pthread_mutex_init(&EventMutex, NULL);
    pthread_cond_init(&Event, NULL);
    pthread_create(&ThreadA, NULL, worker1, NULL);
    pthread_create(&ThreadB, NULL, worker2, NULL);
    //...
    return(0);
}

```

**Table 5-8. Functions of the Attribute Object for the Condition Variable of Type `pthread_cond_t`**

<b>pthread_cond_t Function</b>	<b>Attribute Prototypes</b>	<b>Object #include</b>	<b>Description</b>
<b>&lt;pthread.h&gt;</b>			
<code>int pthread_condattr_init (pthread_condattr_t * attr);</code>			Initializes a condition variable attribute object specified by the parameter <code>attr</code> with default values for all of the attributes defined by the implementation.
<code>int pthread_condattr_destroy (pthread_condattr_t * attr);</code>			Destroys a condition variable attribute object specified by the parameter <code>attr</code> . Can be reinitialized by calling the <code>pthread_condattr_init()</code> function.
<code>int pthread_condattr_ setpshared (pthread_condattr_t * attr, int pshared);</code>			Sets or returns the process-shared attribute of the condition variable attribute object specified by the parameter <code>attr</code> . The <code>pshared</code> parameter contains a value:
<code>int pthread_condattr_ getpshared (const pthread_condattr_t * restrict attr, int *restrict pshared);</code>		<code>PTHREAD_PROCESS_SHARED</code>	Permits a read–write lock to be shared by any threads that have access to the allocated memory of the condition variable even if the threads are in different processes.
		<code>PTHREAD_PROCESS_PRIVATE</code>	The condition variable is shared between threads of the same process as the initialized condition.
<code>int pthread_condattr_ setclock (pthread_condattr_t * attr, clockid_t clock_id);</code>			Sets and returns the clock attribute for the condition variable attribute object specified by the parameter <code>attr</code> . The clock attribute is the clock id of the clock used to measure the timeout service of the <code>pthread_cond_timedwait()</code> function. The default value of
<code>int pthread_condattr_ getclock (const pthread_condattr_t</code>			

<b>pthread_cond_t</b>	<b>Attribute</b>	<b>Object</b>	<b>Description</b>
<b>Function</b>	<b>Prototypes</b>	<b>#include</b>	
		<b>&lt;pthread.h&gt;</b>	

```
* restrict attr,
clockid_t *
restrict clock_id);
```

the clock attribute is the system clock.

In [Example 5.4](#), the FS synchronization relationship is implemented. ThreadA cannot finish until ThreadB starts. ThreadA signals to ThreadB once Number has a value of 50. It can now continue execution until finished. ThreadB cannot start its computation until it gets a signal from ThreadA. ThreadB uses the EventMutex with the condition variable Event. Mutex is used to synchronize write access to the shared data Number. A task can use several mutexes to synchronize different critical sections and synchronize different events.

[Example 5.5](#) contains an implementation of a FF synchronization relationship.

**Example 5.5 FF synchronization relationship between two threads.**

```
//...
float Number;
pthread_t ThreadA,ThreadB;
pthread_mutex_t Mutex,EventMutex;
pthread_cond_t Event;

void *worker1(void *X)
{
    for(int Count = 1;Count < 10;Count++){
        pthread_mutex_lock(&Mutex);
        Number++;
        pthread_mutex_unlock(&Mutex);
        cout << "worker1: number is " << Number << endl;
    }
    pthread_mutex_lock(&EventMutex);
    cout << "worker1 done now waiting " << endl;
    pthread_cond_wait(&Event,&EventMutex);
    pthread_mutex_unlock(&EventMutex);
    return(0);
}

void *worker2(void *X)
{
    for(int Count = 1;Count < 100;Count++){
        pthread_mutex_lock(&Mutex);
        Number = Number * 2;
        pthread_mutex_unlock(&Mutex);
        cout << "worker2: number is " << Number << endl;
    }
    pthread_cond_signal(&Event);
    cout << "worker2 done now signalling " << endl;
    return(0);
}
```

```

int main(int argc, char *argv[])
{
    pthread_mutex_init(&Mutex, NULL);
    pthread_mutex_init(&EventMutex, NULL);
    pthread_cond_init(&Event, NULL);
    pthread_create(&ThreadA, NULL, worker1, NULL);
    pthread_create(&ThreadB, NULL, worker2, NULL);
    //...
    return(0);
}

```

In [Example 5.5](#), ThreadA cannot finish until ThreadB finishes. ThreadA must iterate through the loop 10 times, where ThreadB must iterate through the loop only 100 times. ThreadA will complete its iterations before ThreadB but will wait until ThreadB signals that it is done.

SS and SF can be implemented in a similar manner. These techniques can easily be used to synchronize order of execution between processes.

## 5.4 Synchronization: An Object-Oriented Approach

One of the advantages of object-oriented programming is the protection encapsulation provides for the data component of an object. Encapsulation can provide "object-access policies and usage guidelines" (Hughes & Hughes, 1997) for the user of the object. In the examples presented in this chapter, access policies were the responsibility of the user of the data. With objects and encapsulation, the responsibility has switched from the user of the data to the data itself. This approach creates data, not unlike functions, which are thread safe.

In order to accomplish this, the data (wherever possible) of the multithreaded application should be encapsulated using the C++ class or struct constructs. Then encapsulate the synchronization mechanism such as semaphores, read-write locks, and event mutexes. If the data or synchronization mechanisms are already objects, create an interface class for them. Lastly, combine the data object with the synchronization objects through inheritance or composition, to create data objects that are thread safe. This approach is discussed in detail in [Chapter 11](#).

### Summary

Synchronization can be used to coordinate the order of execution of processes and threads called task synchronization as well as access the shared data called data synchronization. There are four basic task synchronization relationships. A start-to-start relationship means task A cannot start until task B starts. A finish-to-start relationship means task A cannot finish until task B starts. A start-to-finish relationship means task A cannot start until task B finishes. A finish-to-finish (FF) relationship means task A cannot finish until task B finishes. The POSIX standard defines a condition variable of type `pthread_cond_t` that can be used to implement these task synchronization relationships.

The algorithm types of the PRAM model can be used to describe data synchronization. EREW (exclusive read exclusive write) access policy can be implemented with a mutex semaphore. The mutex semaphore protects the critical section by serializing entry into the critical section. Either read access or write access is allowed. The POSIX standard defines a mutex semaphore of type `pthread_mutex_t` that can be used to implement an EREW access policy. Read–write locks can be used to implement the CREW access policy. CREW access policy describes multiple concurrent reads of data but an exclusive write to that data. The POSIX standard defines a read–write lock of type `pthread_rwlock_t`. An object-oriented approach to data synchronization embeds synchronization inside the data object.

## Chapter 6. Adding Parallel Programming Capabilities to C++ Through the PVM

"We have thus divided our problem into two parts. The child-programme and the education process. These two remain very closely connected. We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine, and see how well it learns . . ."

—Alan Turing, Can A Machine Think?

In this Chapter

- [The Classic Parallelism Models Supported by PVM](#)
- [The PVM Library for C++](#)
- [The Basic Mechanics of the PVM](#)
- [Accessing Standard Input \(stdin\) and Standard Output \(stdout\) within PVM Tasks](#)
- [Summary](#)

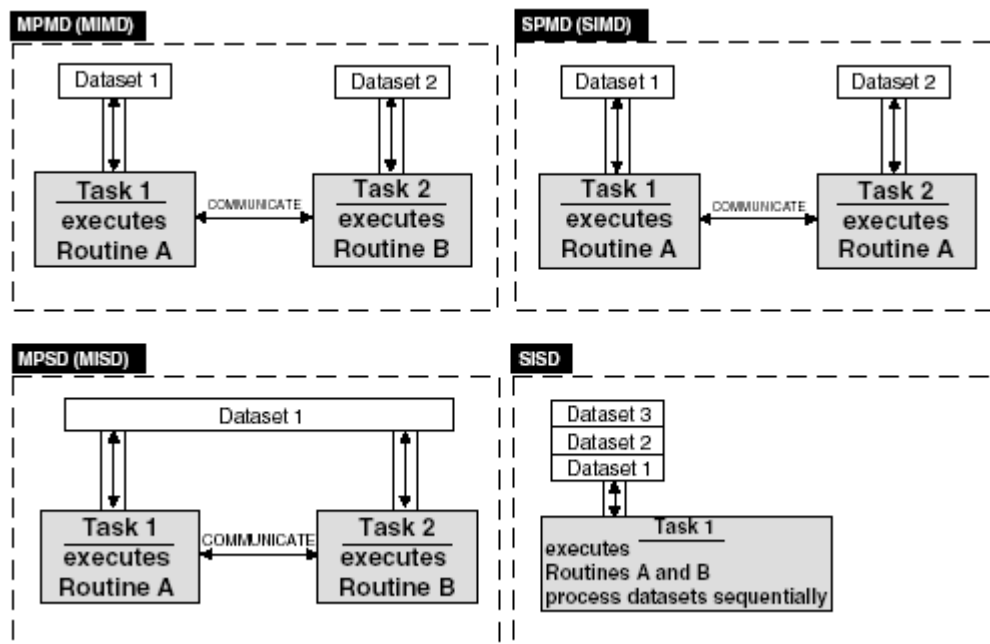
The PVM (Parallel Virtual Machine) is a software system that provides the software developer with the facilities to write and run programs that exploit parallelism. The PVM presents a collection of networked computers to the developer as a single logical machine with parallel capabilities. The collection of computers can all have the same architecture or the collection can consist of computers with different architectures. The PVM can even be connected to computers that fall into the MPP (Massively Parallel Processor) class. Although PVM programs can be developed for a single computer, the real advantages come when there are two or more computers connected.

The PVM supports the message passing model as a means of communication between concurrently executing tasks. An application interacts with the PVM through a library that consists of APIs for process control, sending messages, receiving messages, signaling processes, and so on. A C++ program interfaces with the PVM library in the same way that it interacts with any other function library. While a program that accesses PVM library calls does require certain functions to be called to initialize the environment, there is nothing that forces any particular form or architecture on a C++ program. This means that the C++ programmer can combine PVM capabilities with other styles of C++ programming (e.g., object-oriented, parameterized programming, agent-oriented programming, and structured programming). The use of libraries to provide additional functionality to C++ is considered one of its advantages. Through the use of libraries such as PVM, MPI, or Linda, a C++ developer can use different models of parallelism, whereas other languages are restricted to whatever parallel primitives are built into the language. The PVM library is perhaps the easiest way to add parallel programming capabilities to the C++ language.

## 6.1 The Classic Parallelism Models Supported by PVM

The PVM system supports the MIMD (Multiple Instruction Multiple Data) and SPMD (Single Program Multiple Data) models of parallelism. Actually, SPMD is a variation on the SIMD (Single Instruction Multiple Data) model. The models classify programs by instruction streams and data streams. In the MIMD model, a program consists of two or more concurrently executing instruction streams, each with its own local data stream. Essentially, each processor has its own memory. In the PVM environment the MIMD is considered a distributed memory model, which is in contrast to a shared memory model. In shared memory models each processor can see the same memory locations. In the distributed model memory values must be communicated through message passing. On the other hand, the SPMD model consists of a single program (the same set of instructions) concurrently executing on two or more machines with the program on each machine processing a different data stream. In other words, the same program on each machine is working with different pieces of data. The PVM environment supports both the MIMD and SIMD or a combination of these two models. [Figure 6-1](#) shows the four classic models and where PVM programs are classified.

Figure 6-1. Four classic models of parallelism and the classification of PVM programs.



Notice in [Figure 6-1](#) that the SISD and MISD models are not applicable to the PVM. The SISD model describes a uniprocessor machine and the MISD model has not yet been practically applied. The two models in [Figure 6-1](#) that can be used with PVMs determine how a C++ program interacts with computers. The software developer sees one logical virtual computer as allowing either two or more different concurrently executing tasks, each with access to its own data, or the same task executing as a set of concurrent clones, with each clone accessing some different piece of data. For our purposes the Multiple Instructions and Single Program in [Figure 6-1](#) refer to PVM tasks.

## 6.2 The PVM Library for C++

The PVM functionality is accessed by C++ through a collection of library routines provided by the PVM environment. The routines are typically divided into seven categories:

- Process Management and Control
- Messaging Packing and Sending
- Message Unpacking and Receiving
- Task Signaling
- Message Buffer Management
- Information and Utility Functions
- Group Operations

The library routines are easy to integrate into the C++ environment. The `pvm_` prefix to each function helps to keep the namespace clear. To use the PVM library routines, your programs must include the `pvm3.h` header file and link to `libpvm`. [Programs 6.1](#) and [6.2](#) show how a simple PVM program works. The instructions for compiling and executing [Program 6.1](#) are contained in [Program Profile 6.1](#).

### Program 6.1

```
#include "pvm3.h"
#include <iostream>
#include <string.h>

int main(int argc, char *argv[])
{
    int RetCode, MessageId;
    int PTid, Tid;
    char Message[100];
    float Result[1];
    PTid = pvm_mytid();
    RetCode = pvm_spawn("program6-2", NULL, 0, " ", 1, &Tid);
    if(RetCode == 1){
        MessageId = 1;
        strcpy(Message, "22");
        pvm_initsend(PvmDataDefault);
        pvm_pkstr(Message);
        pvm_send(Tid, MessageId);
        pvm_rcv(Tid, MessageId);
        pvm_upkfloat(Result, 1, 1);
        cout << Result[0] << endl;
        pvm_exit();
        return(0);
    }
    else{
        cerr << "Could not spawn task " << endl;
        pvm_exit();
        return(1);
    }
}
```



## Program Profile 6.1

Program Name

program6-1.cc

Description

Uses `pvm_send` to send a number to another PVM task that is executing ([Program 6.2](#)) and `pvm_rcv` to receive a number from that task.

Libraries Required

libpvm3

Headers Required

<pvm3.h> <iostream> <string.h>

Compile and Link Instructions

[\[View full width\]](#)

```
c++ -o program6-1 -I $PVM_ROOT/include -L $PVM_ROOT/lib/  
➔ $PVM_ARCH -l pvm3
```

Test Environment

Solaris 8, PVM 3.4.3, SuSE Linux 7.1, gcc 2.95.2,

Execution Instructions

```
./program6-1
```

Notes

pvm3d must be running.

[Program 6.1](#) calls eight commonly used PVM routines: `pvm_myid()`, `pvm_spawn()`, `pvm_initsend()`, `pvm_pkstr()`, `pvm_send()`, `pvm_rcv()`, `pvm_upkfloat()`, and `pvm_exit()`. The `pvm_myid()` routine returns the task identifier of the calling process. The PVM system associates a task identifier with each process that it creates. The task identifier is used to send messages between tasks, to receive messages from other tasks, to signal tasks, to interrupt tasks, and so on. Any PVM task may communicate with any other PVM task as long as it has access to the task identifier of the task it wants to communicate with. The `pvm_spawn()` routine is used to start new PVM processes. [Program 6.1](#) uses the `pvm_spawn()` process to start a new process to execute [Program 6.2](#). The task identifier for the new task is returned in the `&Tid` parameter of the `pvm_spawn()` call. The PVM environment uses message buffers to pass data between tasks. Each task can have one or more message buffers. However, only one buffer is considered the active message buffer. Prior to sending each message the `pvm_initsend()` routine is called to prepare or initialize the active message buffer. The `pvm_pkstr()` routine is used to

pack the string contained in the message variable. This packing encodes the string for transport to another task in another process possibly on another machine with a different machine architecture. The PVM environment handles the details of the architecture-to-architecture conversions. The PVM environment requires the use of a packing routine prior to sending and an unpacking routine during receiving to make the message readable by the receiver. However, there is an exception to this, which we will discuss later. The `pvm_send()` and `pvm_rcv()` are used to send and receive messages. The `MessageId` simply identifies which message the caller or sender is working with. Notice in [Program 6.1](#) that the `pvm_send()` and `pvm_receive()` routines contain the task identifier of the task receiving the data and the task identifier of the task sending the data. The `pvm_upkfloat()` routine takes the message it retrieves from the active message buffer and unpacks it into an array of type `float`. [Program 6.1](#) spawns a PVM task to execute [Program 6.2](#).

Notice that [Programs 6.1](#) and [6.2](#) both contain a call to the routine `pvm_exit()`. It's important that this function is called when the PVM processing for a task is finished. Although this routine does not kill the process or stop the process, it does PVM cleanup for the task and disconnects the task from the PVM. Notice that [Programs 6.1](#) and [6.2](#) are self-contained, standalone programs that contain the `main()` function. [Program Profile 6.2](#) has the implementation details for [Program 6.2](#).

### Program 6.2

```
#include "pvm3.h"
#include "stdlib.h"

int main(int argc, char *argv[])
{
    int MessageId, Ptid;
    char Message[100];
    float Num, Result;
    Ptid = pvm_parent();
    MessageId = 1;
    pvm_rcv(Ptid, MessageId);
    pvm_upkstr(Message);
    Num = atof(Message);
    Result = Num / 7.0001;
    pvm_initsend(PvmDataDefault);
    pvm_pkfloat(&Result, 1, 1);
    pvm_send(Ptid, MessageId);
    pvm_exit();
    return(0);
}
```

## Program Profile 6.2

Program Name

program6-2.cc

Description

This program receives a number from its parent process and divides that number by 7. It sends the result to its parent process.

## Libraries Required

libpvm3

## Headers Required

<pvm3.h> <stdlib.h>

## Compile and Link Instructions

[\[View full width\]](#)

```
c++ -o program6-2 -I $PVM_ROOT/include program6-2.cc -L $PVM_ROOT  
/lib/PVM_ARCH -lpvm3
```

## Test Environment

SuSE Linux 7.1 gnu C++ 2.95.2, Solaris 8 Workshop 6, PVM 3.4.3

## Execution Instructions

This program is spawned by [Program 6.1](#).

## Notes

pvmd must be running.

### 6.2.1 Compiling and Linking a C++/PVM Program

Version 3.4.x of the PVM environment packages the routines in a single library, libpvm3.a. To compile a PVM program include the pvm3.h header file and link with libpvm3.a:

```
$ c++ -o mypvm_program -I $PVM_ROOT/include mypvm_program.cc  
-I$PVM_ROOT/lib -lpvm3
```

The \$PVM\_ROOT environment variable points to the PVM installed directory. This command will produce a binary called mypvm\_program.

To execute [Programs 6.1](#) and [6.2](#), you must have the PVM environment properly installed. Three basic methods can be used to execute a PVM program: as a standalone binary, using the PVM console, or using XPVM.

### 6.2.2 Executing a PVM Program as a Standalone

The pvmd program must be started and each host involved in the PVM must have the correctly compiled programs in the appropriate directory. The default directory for the compiled programs (binaries) is:

```
$HOME/pvm3/bin/$PVM_ARCH
```

where the PVM\_ARCH contains the name of the machine's architecture. See [Table 6-2](#) and items 1 and 2 from Section 6.1.5. The binaries should have the proper file permissions set to allow them to be accessed and executed. The pvmd program can be started as:

pvmd &

or:

pvmd hostfile &

where hostfile is a configuration file that has special options to be passed to the pvmd program. See item 5 from section 6.1.5. After the pvmd program has been started on one of the computers involved in the PVM, a PVM program can then be started simply by:

\$MyPvmProgram

If this program spawns any other tasks they will be started automatically.

### 6.2.2.1 Starting PVM Programs Using the PVM Console

To execute the programs using the PVM console, type the following at the PVM console. Start the PVM console by typing:

\$pvm

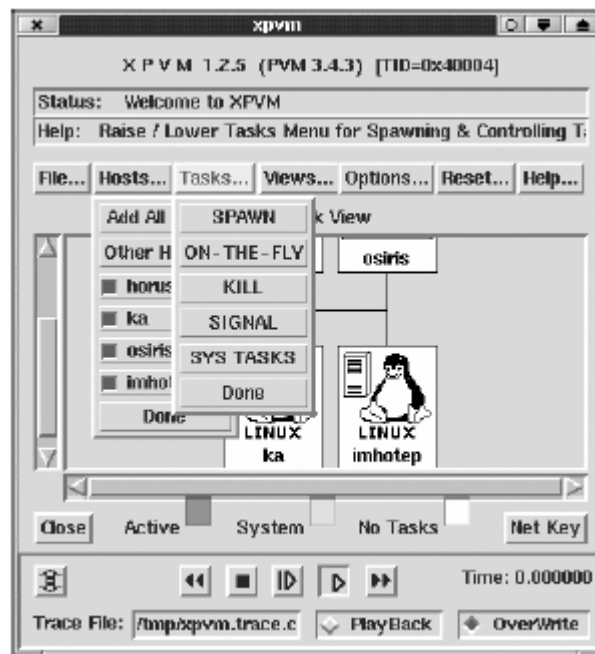
and at the pvm> prompt, type the name of the program to be executed:

pvm> spawn -> MyPvmProgram

### 6.2.2.2 Start PVM Programs Using XPVM

Besides starting the programs using the terminal-based PVM console, XPVM graphical interface for X Windows can be used. [Figure 6-2](#) shows what to type in the tasks dialog of a XPVM session.

Figure 6-2. The XPVM task dialog.



The PVM library does not force any particular structure on a C++ program. The first PVM routine called by a program enrolls that program into the PVM. It is good practice to always call pvm\_exit() for every program that is part of the PVM. If this routine is not called for every PVM task, the system will hang. It is a good rule of thumb to call pvm\_mytid() and pvm\_parent() early in the processing of the

task. [Table 6-1](#) contains the library routines broken down into the seven commonly used categories.

**Table 6-1. Seven Categories of PVM Library Routines**

**Categories of PVM Library Routines Description**

Process Management and Control Routines used to manage and control PVM processes.

Message Packing and Sending Routines used to pack messages into a send buffer and send messages from one PVM process to another.

Message Unpacking and Receiving Routines used to receive messages and unpack the data from the active buffer.

Task Signaling Routines used to signal and notify PVM processes about the occurrence of an event.

Message Management Buffer Routines used to initialize, empty, dispose, and otherwise manage buffers used to receive and send messages between PVM processes.

Information and Utility Functions Routines used to return information about a PVM process and perform other important tasks.

Group Operations Routines used joining, leaving, and otherwise managing processes in a group.

**6.2.3 A PVM Preliminary Requirements Checklist**

In addition to obtaining and properly installing a PVM distribution, there are a few other minor considerations. When the PVM environment is implemented as a network of computers, the following items must be handled before your C++ program can interact with the PVM environment.

Item 1

The environment variable PVM\_ROOT and PVM\_ARCH should be set. The environment variable PVM\_ROOT should be set to the directory where PVM is installed.

Using the Bourne Shell (bash)

Using the C Shell

\$ PVM\_ROOT=/usr/lib/pvm3

setenv PVM\_ROOT /usr/lib/pvm3

\$ export PVM\_ROOT

The PVM\_ARCH environment variable identifies the architecture of the machine. Each machine

involved in the PVM must be identified by architecture. For example, our Ultrasparcs have the designation SUN4SOL2 and our Linux machines have the designation LINUX. [Table 6-2](#) shows the most commonly used architectures for the PVM environment. Check with your distribution of PVM if an appropriate architecture for your machines is not contained in [Table 6-2](#).

[Table 6-2](#) shows the name and machine type associated with the name. Set your PVM\_ARCH environment variable to one of the names in [Table 6-2](#). For instance:

**Table 6-2. Most Commonly Used Architectures for the PVM Environment**

<b>PVM_ARCH</b>	<b>Computer</b>
AFX8	Alliance
ALPHA	DEC Alpha
BAL	Sequent Balance
BFLY	BBN Butterfly TC2000
BSD386	80386/486 PC Running UNIX
CM2	Thinking Machine CM2
CM5	Thinking Machine CM5
CNVX	Convex C-series
CNVXN	Convex C-series
CRAY	C-90, YMP, T3D port available
CRAY2	Cray-2
CRAYSIMP	Cray S-MP
DGAV	Data General Aviion
E88K	Encore 88000
HP300	HP-9000 Model 300

<b>PVM_ARCH</b>	<b>Computer</b>
HPPA	HP-9000 PA-RISC
I860	Intel iPSC/860
IPSC2	Intel iPSC/2 386 Host
KSRI	Kendall Square KSR-1
LINUX	80386/486 PC Running UNIX
MASPAR	Maspar
MIPS	MIPS 4680
NEXT	NeXT
PGON	Intel Paragon
PMAX	DECstation 3100,5100
RS6K	IBM/RS6000
RT	IBM RT
SGI	Silicon Graphics IRIS
SGI5	Silicon Graphics IRIS
SGIMP	SGI Multiprocessor
SUN3	Sun 3
SUN4	Sun 4, SPARCstation
SUN2SOL2	Sun 4, SPARCstation
SUNMP	SPARC Multiprocessor

## PVM\_ARCH

## Computer

SYMM

Sequent Symmetry

TITN

Stardent Titan

U370

IBM 370

UVAX

DEC Licro VAX

Using the Bourne Shell (bash)

Using the C Shell

```
$PVM_ARCH=LINUX
```

```
setenv PVM_ARCH LINUX
```

```
$export PVM_ARCH
```

### Item 2

The binaries (executables) for any programs participating in the PVM have to be either located on all machines involved or accessible by all machines involved in the PVM. In addition to availability, each program must be compiled to work for the architecture it will run on. This means if we have UltraSparcs, PowerPCs, and Intel processors involved in the PVM, then we must have a version of the program compiled for each architecture. That version must be located in a place that the PVM is aware of. The location is often `$HOME/pvm3/bin`. However, the location can be specified in a PVM configuration file usually referred to as the hostfile or `.xpvm_hosts` if the XPVM environment is used. The hostfile would contain an entry such as:

```
ep=/usr/local/pvm3/bin
```

This specifies any user binaries needed by the PVM can be found in the `/usr/local/pvm3/bin` directory.

### Item 3

The user initiating the PVM program must have network access to each machine involved in the PVM. This access is typically rsh or ssh access. See the main pages for more details on the rsh and ssh programs. By default, the PVM accesses each machine using the login name of the user initiating the PVM program or the account name of the machine starting the PVM program. If another account besides the initiating login account is required, an entry must be added to the host file or `.xpvm_hosts`. For example:

```
lo=flashgordon
```

### Item 4

Create a `.rhosts` file on each host listing all the hosts you wish to use. These are the computers that have the potential to be involved in the PVM. Depending on the setting in the `.xpvm_hosts` file or the `pvm_hosts` file, these computers will automatically be added to the PVM when the `pvm` is started.



Computers listed in these files can also be dynamically added to the PVM at runtime.

#### Item 5

Create a `$HOME/.xpvms_hosts` and/or a `$HOME/pvm_hosts` file listing all the hosts you wish to use prepended by an `&`. The `&` means don't automatically add the host. Not using `&` will cause the host to be automatically added. The `pvm_hostfile` is a user-created file. The name is arbitrary. However, `.xpvms_hosts` is the required name when using the XPVM environment. [Figure 6-3](#) shows an example of a PVM hostfile. The same format would be used for the PVM console hostfile or for `.xpvms_hosts`.

**Figure 6-3. An example of a PVM host file.**

```
# Comment lines start with # (empty lines are ignored)
# lines starting with & allow the machine to be loaded into
# the PVM at a later time. If the machine's host name is
# not preceded by & it will be loaded automatically by the
# PVM environment

flavius
marcus
&camblus lo=romulus
&karslus
# The * marks default options for following hosts

* dx=/export/home/fred/pvm3/lib/pvmd
&octavius

# If the computers involved are part of a typical linux
# cluster then the host names as defined in the hosts
# file can be used to include the nodes of the cluster
# with the other nodes in a PVM
```

The primary thing to keep in mind is network access of the user running the PVM program. The owner of the PVM program should have account access to every computer involved in the pool of processors that will be executing parts of the program. This access will use either the `rsh` or `rlogin` commands or `ssh`. The program to be executed must be available on each host and the PVM environment must be aware of what the hosts are and where the binaries will be installed.

### 6.2.4 Combining the C++ Runtime Library and the PVM Library

Since access to the PVM is provided through a collection of library routines, a C++ program treats the PVM as any other library. Keep in mind that each PVM program is a standalone C++ program with its own `main()` function. This means that each PVM program has its own address space. When a PVM task is spawned, a new process is created. Each PVM program will have its own process and process id. The PVM processes are visible to the `ps` utility. Although two or more PVM tasks may be working together to solve some problem, they will have their own copies of the C++ runtime library. Each program has its own `iostream`, `template` library, `algorithms`, and so on. The scope of global C++ variables do not cross address space. This means global variables in one PVM task will be invisible to the other PVM tasks involved in the processing. Message passing is used to communicate between these separate tasks. Notice that this is in contrast to multithreaded programs where threads share the same address space and may communicate through parameter passing and global variables. If the PVM programs are executing on a single computer that has multiple processors, then the programs may share a file system and can use pipes, `fifos`, shared memory, and files as additional means to communicate. While message passing is the premier method of communicating between PVM tasks, nothing prevents the use of shared file systems, clipboards, or even command-line arguments as supplemental methods of communication between tasks. The PVM library adds to rather than restricts the capabilities of the C++ runtime library.

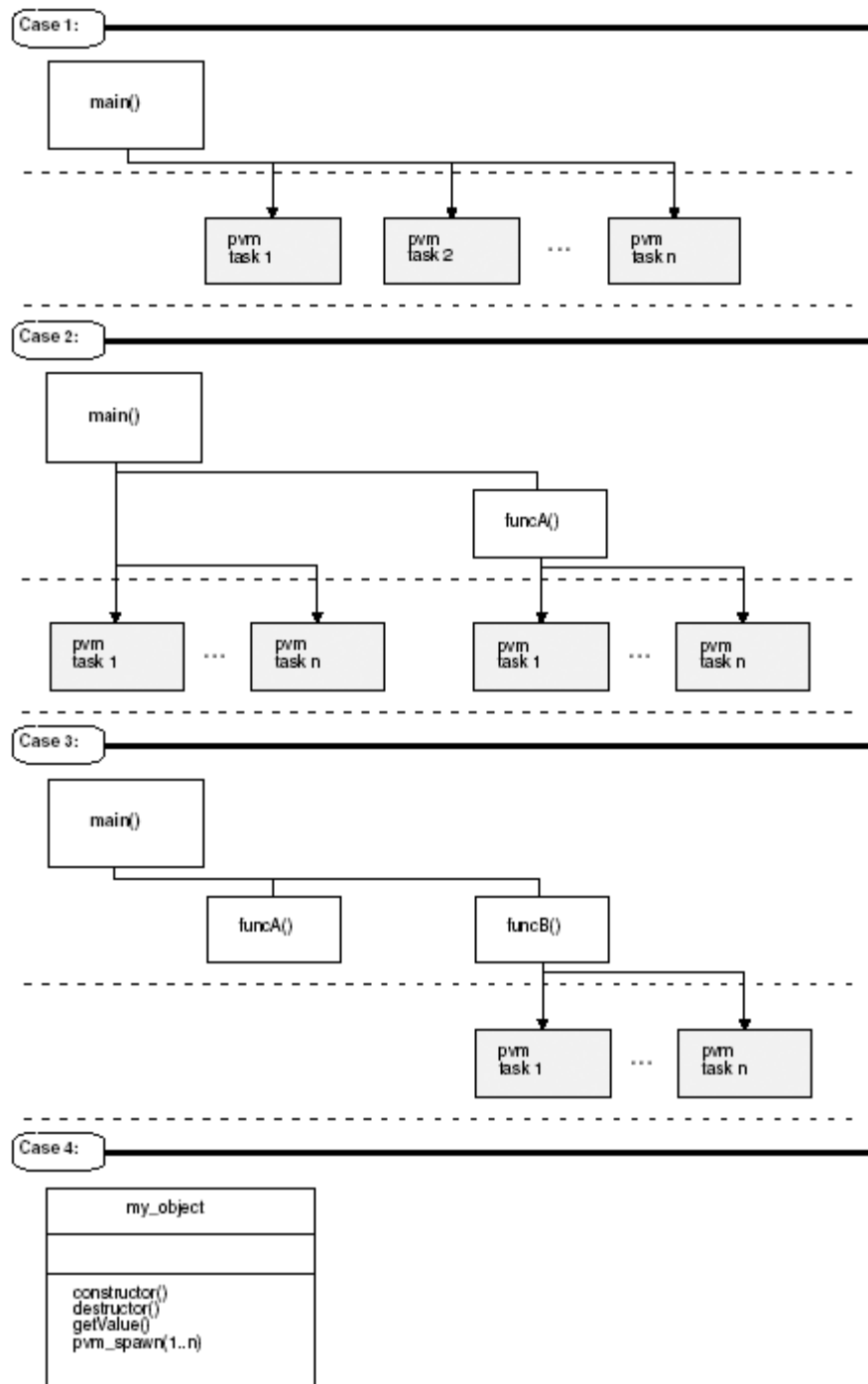
## 6.2.5 Approaches to Using PVM Tasks

The work a C++ program performs can be distributed between functions, objects, or combinations of functions and objects. The units of work in a program usually fall into logical categories: input/output, user interface, database processing, signal processing, error handling, numerical computation, and so on. Also, we try to keep user interface code separated from file processing code and printing routine code separated from the numerical computation code. Therefore, not only do we divide up the work our program does between functions or objects, we try to keep categories of functionality together. These logical groupings are organized into libraries, modules, object patterns, components, and frameworks. We maintain this type of organization when introducing PVM tasks into a C++ program. We can arrive at the WBS (Work Breakdown Structures) using either a bottom-up or top-down approach. In either case, the parallelism should naturally fit within the work that a function, module, or object has to do.

It is not a good idea to attempt to force parallelism in a program. Forced parallelism produces awkward program architectures that are hard to understand by making them hard to maintain and often hard to determine program correctness. So when a program uses PVM tasks, they should be a result of the natural division within the program. Each PVM task should be traceable to one of the categories of work within the program. For instance, if we have an application that has NLP (Natural Language Processing) and TTS (Text to Speech) processing as part of its user interface and inferencing as part of its data retrieval, then the parallelism that is natural within the NLP component should be represented as tasks within the NLP module or object that is responsible for NLP. Likewise, the parallelism within the inferencing component should be represented as tasks within the data retrieval module or the object or framework that is responsible for data retrieval. That is, we identify PVM tasks where they logically fit within the work that the program is doing as opposed to dividing the work the program does into a set of generic PVM tasks.

The notion of logic first, parallelism second, has several implications for C++ programs. It means that we might spawn PVM tasks from the main() function. We might spawn PVM tasks from subroutines called from main() or from other subroutines. We might spawn PVM tasks from within methods belonging to objects. Where we spawn the tasks depends on the concurrency requirements of the function, module, or object that is performing the work. The PVM tasks generally fall into two categories: SPMD (a derivative of SIMD) and MPMD (a derivative of MIMD). In the SPMD model, the tasks will execute the same set of instructions but on different pieces of data. In the MPMD model, each task executes different instructions on different data. Whether we are using the SPMD model or the MPMD model, the spawning of the task should be from the relevant areas of the program. [Figure 6-4](#) shows some possible configurations for spawning PVM tasks.

Figure 6-4. Some possible configurations for spawning PVM tasks.



### 6.2.5.1 Using the SPMD (SIMD) Model with PVM and C++

In [Figure 6-4](#), Case 1 represents the situation where the function `main()` spawns from 1 to N tasks where each task performs the same set of instructions but on different data sets. There are several options for implementing this scenario. [Example 6.1](#) shows `main()` using the `pvm_spawn` routine.

**Example 6.1 Calling the pvm\_spawn routine from main().**

```
int main(int argc, char *argv[])
{
    int TaskId[10];
    int TaskId2[5];
    pvm_spawn("set_combination",NULL,0," ",10,TaskId); // 1rst Spawn
    pvm_spawn("set_combination",argv,0," ",5,TaskId2); // 2nd Spawn
    //...
}
```

In [Example 6.1](#), the first spawn creates 10 tasks. Each task will execute the same set of instructions contained in the set\_combination program. The TaskId array will contain the task identifiers for the PVM tasks if the spawn was successful. Once the program in [Example 6.1](#) has the TaskIds, then it can use the pvm\_send() routines to send specific data for each program to work on. This is possible because the pvm\_send() routine contains the task identifier of the receiving task. The second spawn in [Example 6.1](#) creates five tasks but in this case it passes each task information through the argv parameter. This is an additional method to pass information to tasks during startup. This is another way for a child task to uniquely identify itself by using values it receives in the argv parameter. In [Example 6.2](#), the main() function uses multiple calls to pvm\_spawn() to create N tasks as opposed to a single call.

**Example 6.2 Using multiple calls to pvm\_spawn from main().**

```
int main(int argc, char *argv[])
{
    int Task1;
    int Task2;
    int Task3;
    //...
    pvm_spawn("set_combination",NULL,1,"host1",1,&Task1);
    pvm_spawn("set_combination",argv,1,"host2",1,&Task2);
    pvm_spawn("set_combination",argv++,1,"host 3",1,&Task3);
    //...
}
```

The approach used in [Example 6.2](#) can be used when you want the tasks to execute on specific computers. This is one of the advantages of the PVM environment. A program can take advantage of some particular resource on a particular computer, for example, special math processor, graphics processor, or MPP capabilities. Notice in [Example 6.2](#) each host is executing the same set of instructions but each host received a different command-line argument. Case 2 in [Figure 6-4](#) represents the scenario where the main() function does not spawn the PVM tasks. In this scenario the PVM tasks are logically related to funcB() and therefore funcB() spawns the tasks. The main() function and funcA() don't need to know anything about the PVM tasks so there is no reason to put any of the PVM housekeeping code in those functions. Case 3 in [Figure 6-4](#) represents the scenario where the main() function and other functions in the program have natural parallelism. In this case the other function is funcA(). Also the PVM tasks executed by main() and the PVM tasks executed by funcA() execute different code. Although the tasks that main() spawns execute identical code and the tasks that funcA() spawns executes identical code, the two sets of tasks are different. This illustrates that a C++ program may use collections of tasks to solve different problems simultaneously. There is no reason that the program has to be restricted to one problem at a time. In Case 4 from [Figure 6-4](#), the parallelism is contained within an object, therefore one of the object's methods spawns the PVM tasks. Here, the logical place to initiate the parallelism was within a class as opposed to some free-floating function.

As in the other cases, the PVM tasks spawned in Case 4 all execute the same instructions but with

different data. This SPMD (Single Program Multiple Data) method is a commonly used technique for parallelization of certain kinds of problem solving. The fact that C++ has support for objects and generic programming using templates makes C++ a particularly powerful choice for this kind of programming. The objects and templates allow the C++ programmer to represent very general and flexible solutions to entire classes of problems with a single piece of code. This single piece of code fits in nicely with the SPMD model of parallelism. The notion of a class extends the SPMD model so that an entire class of problems can be solved. The templates allow the class of problems to be solved for virtually any data type. So although each task in the SPMD model is executing the same piece of code, it might be for an object or any of its descendants and it might be for different data types (different objects!). For example, [Example 6.1](#) uses four PVM tasks to generate four sets in which each has  $C(n,r)$  elements:  $C(24,9)$ ,  $C(24,12)$ ,  $C(7,4)$ , and  $C(7,3)$ . Specifically, [Example 6.3](#) enumerates the combinations of a set of 24 colors taken 9 and 12 at a time. It also enumerates the combinations of a set of 7 floating point numbers taken 4 at a time and 3 at a time. For an explanation of the notation  $C(n,r)$ , see [Sidebar 6.1](#).

#### Example 6.3 Creating combinations of sets.

```
int main(int argc,char *argv[])
{
    int RetCode,TaskId[4];
    RetCode = pvm_spawn("pvm_generic_combination",NULL,0," ",4,TaskId);
    if(RetCode == 4){
        colorCombinations(TaskId[0],9);
        colorCombinations(TaskId[1],12);
        numericCombinations(TaskId[2],4);
        numericCombinations(TaskId[3],3);
        saveResult(TaskId[0]);
        saveResult(TaskId[1]);
        saveResult(TaskId[2]);
        saveResult(TaskId[3]);
        pvm_exit();
    }
    else{
        cerr << "Error Spawning ChildProcess" << endl;
        pvm_exit();
    }
    return(0);
}
```

Notice in [Example 6.3](#) we spawn four PVM tasks:

```
pvm_spawn("pvm_generic_combination",NULL,0," ",4,TaskId);
```

Each task will execute the program named `pvm_generic_combination`. The `NULL` argument in our `pvm_spawn` call means that we are not passing any options via the `argv[]` parameter. The `0` in our `pvm_spawn` call means we don't care which computer the tasks execute on. `TaskId` is an array of four integers and will contain the task identifiers for each of the PVM tasks spawned if the call is successful. Notice in [Example 6.3](#) we call `colorCombinations()` and `numericCombinations()`. These two functions assign the PVM tasks work. [Example 6.4](#) contains the function definition for `colorCombinations()`.

**Example 6.4 Definition of the colorCombinations() function.**

```
void colorCombinations(int TaskId,int Choices)
{
    int MessageId =1;
    char *Buffer;
    int Size;
    int N;
    string Source("blue purple green red yellow orange
                 silver gray ");
    Source.append("pink black white brown light_green
                 aqua beige cyan ");
    Source.append("olive azure magenta plum orchid violet
                 maroon lavender");
    Source.append("\n");
    Buffer = new char[(Source.size() + 100)];
    strcpy(Buffer,Source.c_str());
    N = pvm_initsend(PvmDataDefault);
    pvm_pkint(&Choices,1,1);
    pvm_send(TaskId,MessageId);
    N = pvm_initsend(PvmDataDefault);
    pvm_pkbyte(Buffer,strlen(Buffer),1);
    pvm_send(TaskId,MessageId);
    delete Buffer;
}
```

Notice in [Example 6.3](#) there are two calls to colorCombinations(). Each call assigns a PVM task a different number of color combinations to enumerate:  $C(24,9)$  and  $C(24,12)$ . The first PVM task will produce 1,307,504 color combinations and the second task will produce 2,704,156 color combinations. The program named in the pvm\_spawn() call does all the work. Each color is represented by a string. Therefore, when the pvm\_generic\_combination program is producing combinations it does so using a set of strings as the input. This is in contrast to the numericCombinations() function shown in [Example 6.5](#). The code in [Example 6.3](#) makes two calls to the numericCombinations() function. The first generates  $C(7,4)$  combinations and the second generates  $C(7,3)$  combinations.

**Example 6.5 Using PVM tasks to produce numeric combinations.**

```
void numericCombinations(int TaskId,int Choices)
{
    int MessageId = 2;
    int N;

    double ImportantNumbers[7] = {3.00e+8,6.67e-11,1.99e+30,
                                  1.67e-27,6.023e+23,6.63e-34,
                                  3.14159265359};

    N = pvm_initsend(PvmDataDefault);
    pvm_pkint(&Choices,1,1);
    pvm_send(TaskId,MessageId);
    N = pvm_initsend(PvmDataDefault);
    pvm_pkdouble(ImportantNumbers,5,1);
    pvm_send(TaskId,MessageId);
}
```

In the numericCombinations() function in [Example 6.4](#), the PVM task is sent an array of floating point numbers as opposed to an array of bytes representing strings. So the colorCombinations() function sends its data to the PVM tasks using:

```
pvm_pkbyte(Buffer, strlen(Buffer), 1);
pvm_send(TaskId, MessageId);
```

The numericCombination() function sends its data to the PVM tasks using:

```
pvm_pkdouble(ImportantNumbers, 5, 1);
pvm_send(TaskId, MessageId);
```

The colorCombinations() function in [Example 6.4](#) builds a string of colors and then copies that string of colors into an array of char called Buffer. The array of char is then packed and sent to the PVM task using the pvm\_pkbyte() and pvm\_send() functions. The numericCombinations() function in [Example 6.5](#) creates an array of doubles and sends it to the PVM task using the pvm\_pkdouble() and pvm\_send() functions. One function sends a character array; the other function sends an array of doubles. In both cases the PVM tasks are executing the same program pvm\_generic\_combination. This is where the advantage of using C++ templates and genericity comes in. The same tasks are able to do work not only with different data but on different data types without a code change. The template facility in C++ helps to make the SPMD model more flexible and efficient. The pvm\_generic\_combination program is almost unaware of what data types it will be working with. The use of C++ container classes allows it to generate combinations of any vector<T> of objects. The pvm\_generic\_combination program does know that it will be working with two data types. [Example 6.6](#) shows a section of code from the pvm\_generic\_combination program.

**Example 6.6 Using the MessageId tag to distinguish data types.**

```
pvm_bufinfo(N, &NumBytes, &MessageId, &Ptid);
if(MessageId == 1){
    vector<string> Source;
    Buf = new char[NumBytes];
    pvm_upkbyte(Buf, NumBytes, 1);
    stringstream Buffer;
    Buffer << Buf << ends;
    while(Buffer.good())
    {
        Buffer >> Color;
        if(!Buffer.eof()){
            Source.push_back(Color);
        }
    }
    generateCombinations<string>(Source, Ptid, Value);
    delete Buf;
}
if(MessageId == 2){
    vector<double> Source;
    double *ImportantNumber;
    NumBytes = NumBytes / sizeof(double);
    ImportantNumber = new double[NumBytes];
    pvm_upkdouble(ImportantNumber, NumBytes, 1);
    copy(ImportantNumber, ImportantNumber + (NumBytes + 1),
        inserter(Source, Source.begin()));
    generateCombinations<double>(Source, Ptid, Value);
    delete ImportantNumber;
}
}
```

Here we use the MessageId tag to distinguish which data type we are working with. But in C++ we can do better. If the MessageId tag contains a 1, then we are working with strings. Therefore, we make the

declaration:

```
vector<string> Source;
```

If the MessageId tag contains a 2, then we know we are working with floating point numbers, and we make the declaration:

```
vector<double> Source;
```

Once we declare what type of data the vector source will contain, the rest of the function in the `pvm_generic_combination` is generalized. Notice in [Example 6.6](#) that each `if()` statement calls the `generateCombinations()` function. This `generateCombinations()` function is a template function. This template architecture helps us to achieve the genericity that will extend the SPMD and the MPMD scenarios for our PVM programs. We will come back to the discussion of our `pvm_generic_combination` program after we present the basic mechanics of the PVM environment. It is important to note that C++ container classes, stream classes, and template algorithms add flexibility to PVM programming that cannot be easily implemented in other PVM environments. This flexibility creates opportunities for sophisticated yet elegant parallel architectures.

#### 6.2.5.2 Using the MPMD (MIMD) Model with PVM and C++

Whereas the SPMD model uses the `pvm_spawn()` function to create some number of tasks executing the same program but on potentially different data or resources, the MPMD model will use the `pvm_spawn()` function to create tasks that are executing different programs each with their own data sets. [Example 6.7](#) shows how a single C++ program could implement a MPMD model of computation using PVM calls.

#### Example 6.7 Using PVM to implement the MPMD model of computation.

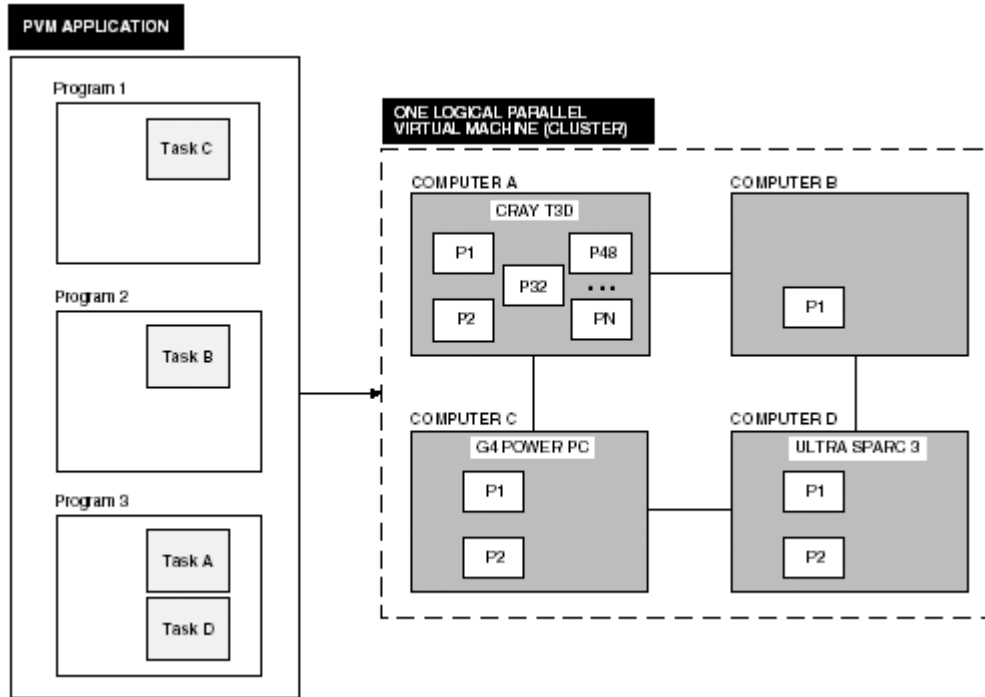
```
int main(int argc, char *argv[])
{
    int Task1[20];
    int Task2[50];
    int Task3[30];
    //...
    pvm_spawn("pvm_generic_combination",NULL,1,"host1",20,Task1);
    pvm_spawn("generate_plans",argv,0," ",50,Task2);
    pvm_spawn("agent_filters",argv++,1,"host 3",30,&Task3);
    //...
}
```

The code in [Example 6.7](#) creates 100 tasks. The first 20 tasks are generating combinations. The next 50 tasks are generating plans from the combinations as the combinations are being created. The last 30 tasks are filtering the best plans from the set of plans being generated by the set of 50 tasks. This is in contrast to the SPMD model, where all of the programs spawned by the `pvm_spawn()` function were the same. Here, we have `pvm_generic_combination`, `generate_plans`, and `agent_filters` performing the work of the PVM tasks. They are all executing concurrently. They each have their own set of data; although they are working with transformations of the data. The `pvm_generic_combination` transforms its input into something that `generate_plans` can use. The `generate_plans` program transforms its input into something that `agent_filters` can use. Obviously these tasks will send messages to each other. The messages will represent input and control information between the processes. Also notice in [Example 6.7](#) that we used the `pvm_spawn()` routine to allocate 20 `pvm_generic_combination` on a computer named `host1`. The `generate_plans` task was allocated to 50 anonymous processors, but each of the 50 tasks received the same command-line argument through the `argv` parameter. The `agent_filters` tasks



were also directed to a particular computer. In this case, the computer was host 3, and each task received the same command-line argument through the argv parameter. This emphasizes the flexibility and power of the PVM library. [Figure 6-5](#) shows some options available for MPMD configurations using the PVM environment.

**Figure 6-5.** Some options available for MPMD configurations using the PVM environment.



We can take advantage of particular resources of particular computers if so desired. We can use arbitrary anonymous computers in other cases. In addition, we can assign different work to different tasks simultaneously. In [Figure 6-5](#) Computer A is a MPP (Massively Parallel Processor) computer, and Computer B has a number of specialized numeric processors. Also notice that the PVM in [Figure 6-5](#) consists of PowerPCs, Sparcs, Crays, and so on. In some cases we don't care what specific capabilities of the computers in a PVM are, but in other cases we do. The `pvm_spawn()` routine allows the C++ programmer to use the anonymous approach by simply not specifying which computer to create the tasks on. On the other hand, if there is something special about some member of the PVM, then that feature can be exploited by specifying the particular member using `pvm_spawn()`.

## S 6.1. Combination Notation

Suppose we wish to choose a team of eight programmers from a pool of 24 candidates. How many different teams of eight programmers could we come up with? One of the results that follow from the Fundamental Principle of Counting tells us there are 735,471 different teams consisting of eight programmers that can be selected from a pool of 24. The notation  $C(n,r)$  is read the number of combination of  $n$  choose  $r$ . That is, the number of choices taken  $r$  at a time from  $n$  items.  $C(n,r)$  is calculated by the formula:

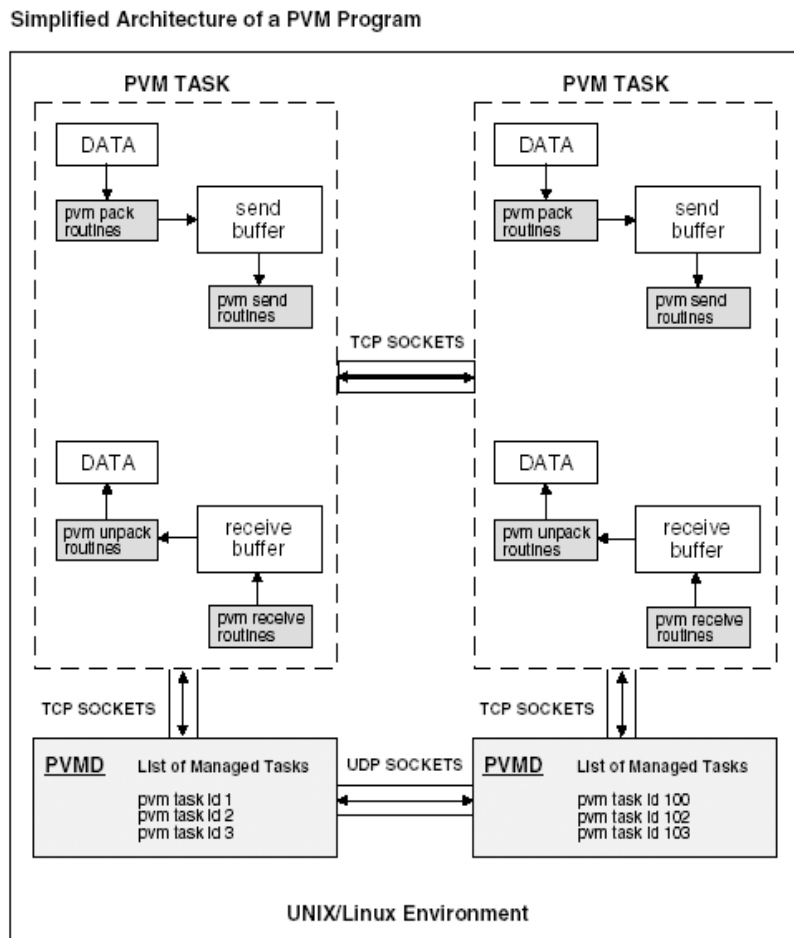
$$\frac{n!}{r!(n-r)!}$$

When we have a set that represents a combination, for example, {a,b,c} would be considered the same as the set {b,a,c}, or {c,b,a}. That is, we don't care about the order of the members in the set; we are only concerned about the members in the set. Many parallel programs, search algorithms, heuristics, and artificial intelligence-based programs have to deal with large sets of combinations and their close relative, permutations.

### 6.3 The Basic Mechanics of the PVM

The PVM environment consists of two components: the PVM daemon (pvmd) and the pvmd library. One pvmd daemon runs on each host computer in the virtual machine. The pvmd serves as a message router and controller. A pvmd is used to start additional pvmds. Each pvmd manages the lists of PVM tasks on its host machine. The pvmd performs process control, some minimal authentication, and fault tolerance. Usually the first pvmd is started manually. This pvmd then starts the other pvmds. Only the original pvmd may start additional pvmds. Only the original pvmd may unconditionally stop another pvmd. The pvmd library contains the routines that allow one PVM task to interact with other PVM tasks. The library also contains routines that allow the PVM task to communicate with its pvmd. [Figure 6-6](#) shows the basic architecture of the PVM environment.

Figure 6-6. Basic architecture of the PVM environment.



The PVM environment will consist of two or more PVM tasks. Each task will contain one or more send

buffers. However, only one send buffer may be active at a time. This is called the active send buffer. Each task has an active receive buffer. Notice in [Figure 6-6](#) that communication between PVM tasks is actually accomplished using TCP sockets. The `pvm_send()` routines make socket access transparent. The programmer does not access the TCP socket calls directly. [Figure 6-6](#) also shows PVM tasks communicating to their pvmds using TCP sockets and pvmds communicating between themselves using UDP sockets. Again, the socket calls are performed by the PVM routines. The programmer does not have to do low-level socket programming. The PVM routines we use in this book fit into four categories:

- Process Management and Control
- Messaging Packing and Sending
- Message Unpacking and Receiving
- Message Buffer Management

While there are other categories of PVM routines, such as the Information and Utility Functions and the Group Operations, we focus on the message processing and process management routines. We will discuss any other routines in the context of the programs in which they are used.

### 6.3.1 Process Management and Control Routines

There are six commonly used process management and control routines.

#### Synopsis

```
#include "pvm3.h"

int pvm_spawn(char *task, char **argv, int flag, char *location,
              int ntask,int *taskids);
int pvm_kill(int taskid);
int pvm_exit(void);
int pvm_addhosts(char **hosts,int nhosts,int *status);
int pvm_delhosts(char **hosts,int nhosts,int *status);
int pvm_halt(void);
```

The `pvm_spawn()` routine is used to create new PVM tasks. The routine can specify how many tasks to create, where to create the tasks, and arguments to be passed to each task. For example:

```
pvm_spawn("agent_filters",argv++,1,"host 3",30,&Task3);
```

The task parameter should contain the name of the program that the `pvm_spawn()` is to execute. Since the program that is executed by the `pvm_spawn()` routine is a standalone program, command-line arguments may be required. The `argv` parameter is used to pass any command-line arguments to the program. The location parameter specifies which host the task is to be executed on. The `taskids` parameter will contain either the task identifiers for the spawned tasks or status codes representing any error conditions that might have been created during the spawn process. The `ntasks` parameter specifies how many instances of the task to create. The `pvm_kill()` routine is used to kill tasks other than the calling task. The `taskid` passed to `pvm_kill()` can reference any other user-defined task in the PVM. This routine works by sending the SIGTERM signal to the PVM task to be killed. The `pvm_exit()` routine is used to cause the calling task to be removed from the PVM. While the task can be removed from the PVM, the process that the task belonged to may continue to execute. Keep in mind that a task executing PVM calls may have other work to perform that is not related to the PVM. The `pvm_exit()` routine

should be called by any task that no longer has work relevant to the PVM processing. The `pvm_addhosts()` allows the caller to dynamically add more computers to an existing PVM. Typically, the `pvm_addhosts()` is called with a list of one or more hostnames:

```
int Status[3];
char *Hosts[] = {"porthos", "dartagnan", "athos"};
pvm_addhosts("porthose", 1, &Status);
```

```
//... or ...
```

```
pvm_addhosts(Hosts, 3, Status);
```

The `Hosts` parameter will usually contain the names of one or more hosts listed in the `.rhosts` file or the `.xpvm_hosts` file. The `nhost` parameter will contain the number of hosts to be added to the PVM, and the `status` parameter will contain a value = to `nhosts` if the call was successful. If the call was not able to add any hosts, the return value will be less than 1. If the call was only partially successful, the return value will represent the number of hosts added. Likewise, the `pvm_delhosts()` allows the caller to dynamically remove one or more computers from an existing PVM. The `hosts` parameter will contain a list of one or more hosts. The `nhosts` parameter will contain the number of hosts to be removed. For instance:

```
pvm_delhosts("dartagnan", 1);
```

causes the computer with hostname `dartagnan` to be removed from the PVM environment. The `pvm_addhosts()` and `pvm_delhosts()` may be called during runtime. This allows the programmer to have a dynamically sizeable PVM. Any PVM task running on a host computer that is deleted from the PVM will be killed. If there are any `pvm`s running on the computers that are deleted from the PVM, the `pvm`s will be stopped also. If a host fails for some reason, the PVM environment will automatically delete the host. The return values for `pvm_delhosts` are the same as they are `pvm_addhosts()`. The `pvm_halt()` routine shuts down the entire PVM system. All tasks and `pvm`s are stopped.

### 6.3.2 Message Packing and Sending

Geist, Beguelin, and colleagues state the message model of the PVM environment accordingly:

PVM daemons and tasks can compose and send messages of arbitrary lengths containing typed data. The data can be converted using XDR<sup>1</sup> when passing between hosts with incompatible data formats. Messages are tagged at send time with a user-defined integer code and can be selected for receipt by source address or tag. The sender of a message does not wait for an acknowledgement from the receiver, but continues as soon as the message has been handed to the network and the message buffer can be safely deleted or reused. Messages are buffered at the receiving end until received. PVM reliably delivers messages, provided the destination exists. Message order from each sender to each receiver in the system is preserved; if one entity sends several messages to another, they will be received in the same order.

The PVM library consists of a family of routines used to pack the various data types into a send buffer. There are pack routines for character arrays, doubles, floats, ints, longs, bytes, and so on. [Table 6-3](#) shows the list of `pvm` routines by type.

**Table 6-3. pvmpk Routines**

### **Message Packing Functions**

bytes

```
int pvm_pkbyte(char *cp, int count, int std);
```

complex/double complex

```
int pvm_pkcplx(float *xp, int count, int std);  
int pvm_pkdcplx(double *zp, int count, int std);
```

double

```
int pvm_pkdouble(double *dp, int count, int std);
```

float

```
int pvm_pkfloat(float *fp, int count, int std);
```

int

```
int pvm_pkint(int *np, int count, int std);
```

long

```
int pvm_pklng(long *np, int count, int std);
```

short

```
int pvm_pkshort(short *np, int count, int std);
```

string

```
int pvm_pkstr(char *cp);
```

Each of the pack routines in [Table 6-3](#) are used to store an array of data in the send buffer. Notice in [Figure 6-6](#) that each PVM task will have at least one send and receive buffer. Each of the pack routines takes a pointer to an array of the appropriate data type. Every pack routine except for `pvm_pkstr()` takes the total number of items to be stored in the array (not the number of bytes!). The `pvm_pkstr()` routine assumes the character array it is working with will be NULL terminated. Each pack routine except the `pvm_pkstr()` has as the last parameter a value that represents how to traverse the source array as items are selected to be packed into the send buffer. The parameter is often referred to as the stride. For

instance, if the stride is four, then every fourth element will be selected from the source to be stored in the send buffer. It is important to note that the `pvm_initsend()` routine should be used prior to sending each message. The `pvm_initsend()` routine clears the buffer and prepares it to send the next message. The `pvm_initsend()` routine prepares the buffer to send the message in one of three formats: XDR, Raw, or In Place.

XDR (External Data Representation) is a standard used to describe and encode data. Keep in mind that the hosts involved in a PVM can be different machine types. For instance, a PVM might consist of Sun, Macintosh, Crays, and AMD machines. These machines might have different word sizes and may store data types differently. In some instance the bit ordering might be different from one machine to another. The XDR standard is used to allow the machines to exchange data in an architecture-independent way. The Raw format is used to send the data in the native format of the sending machine. No special encoding is used. The in place format really does not pack the data in the send buffer. Instead, only pointers to the data and size of the data is sent. The receiving task copies the data directly. These three types of encoding are represented by three constants in the PVM library:

<code>PvmDataDefault</code>	XDR
<code>PvmDataRaw</code>	No special encoding
<code>PvmDataInPlace</code>	Only pointers and sizes copied to send buffer

For example:

```
int BufferId;
BufferId = pvm_initsend(PvmDataRaw);
//...
```

specifies that data be packed into the send buffer as is, that is, with no special encoding. If the `pvm_initsend()` call is successful, it will return the number of the send buffer in `BufferId`. It is important to remember that although only one send buffer can be active at a time, a PVM task can have multiple send buffers. Each buffer will have a number associated with it. The PVM library defines several send routines.

## Synopsis

```
#include "pvm3.h"

int pvm_send(int taskid, int messageid);
int pvm_psend(int taskid, int messageid, char *buffer, int len,
              int datatype);
int pvm_mcast(int *taskid, int ntask, int messageid);
```

In each of these routines, `taskid` is the task identifier of the PVM task that receives the message. In the case of `pvm_mcast()`, the `taskid` refers to a collection of tasks represented by the task identifiers passed in the array `*taskid`. The `messageid` parameter specifies which message to send. The message identifiers are integers and are user-defined. They are used by the sender and receiver to identify which message will be waited on by the receiver and which message will be sent by the sender. For example:

```

pvm_bufinfo(N,&NumBytes,&MessageId,&Ptid);

//...

switch(MessageId)
{
    case 1 : // do something
            break;

    case 2 : // do something else
            break
            //...
}

```

In this case, the `pvm_bufinfo()` routine is used to get information about the last message received by receive buffer N. We can get the number of bytes, the messageid, and who sent the message. Once we get the messageid we can execute the appropriate logic. The `pvm_send()` routine performs a pseudo-blocking send to the specified task, that is, the task only blocks as long as it takes to make sure that the message has been properly sent. The task does not wait for the receiver to actually receive the task. The `pvm_psend()` routine sends the message directly to the specified task. Notice that the `pvm_psend()` routine has a buffer parameter used to contain the message to be sent. The `pvm_mcast()` is used to send a message to multiple tasks simultaneously. The arguments for the `pvm_mcast()` will include an array of taskids that receives the message, the number of tasks involved in the multicast, and the messageid to identify the message sent. [Figure 6-6](#) shows that each PVM task has its own send buffer. The buffer exists just long enough for the message to be guaranteed to be on its way.

With the exception of control messages, the meaning of the messages between any two PVM tasks is application defined, that is, the sending and the receiving task must have a predefined use for each message. The messages are asynchronous, of arbitrary data types, and of arbitrary length. This provides for maximum flexibility within the application. The counterparts to the `pvm_send` messages are the PVM receive messages. There are five important functions in the receive family of routines.

## Synopsis

```

#include "pvm3.h"

int pvm_rcv(int taskid, int messageid);
int pvm_nrcv(int taskid, int messageid);
int pvm_prcv(int taskid, int messageid, char *buffer,
            int size, int type, int sender, int messageid,
            int messageid);
int pvm_trecv(int taskid, int messageid,
            struct timeval *timeout);
int pvm_probe(int taskid, int messageid);

```

The `pvm_rcv()` routine is used to receive messages from other PVM tasks. This routine creates a new active buffer that will contain the message received. The taskid parameter specifies the task identifier of the sending task. The messageid parameter identifies the message that is being sent from the sender. Keep in mind that a task may send multiple messages, each with different or the same messageid. If the taskid = -1, then the `pvm_rcv()` routine will accept a message from any task. If the messageid parameter = -1, then the routine will accept any message. The `pvm_rcv()` routine return value will be the buffer id of the new active buffer if the call is successful and will be a value < 0 if an error has occurred. When a task calls the `pvm_rcv()` routine will block and wait until the message has been received. After the message is received, it is retrieved from the active message buffer using one of the

unpack routines. For instance:

```
//...
float Value[10];
pvm_recv(400002,2);
pvm_unpackfloat(400002, Value,1);
cout << Value..
```

The `pvm_recv()` routine causes this code to wait on a message from a task identified as 400002. The messageid received from 400002 must be 2 before the routine unblocks. The unpack routine is then used to retrieve the array of floats. Whereas the `pvm_recv()` routine causes the task to wait until it receives a message, the `pvm_nrecv()` routine is a nonblocking receive. If the appropriate message has not arrived, the `pvm_nrecv()` routine will immediately return. If the message has arrived, the `pvm_nrecv()` will return immediately and the active buffer will contain the message. If an error condition occurs, then `pvm_nrecv()` will return a value  $< 0$ . If the message has not arrived, the routine returns 0. If the message has arrived, the number for the new active buffer will be returned. The taskid parameter will contain the task identifier for the sending task. The messageid parameter will contain a user-defined message id. If the taskid = -1, then the `pvm_nrecv()` routine will accept a message from any task. If the messageid = -1, or then the routine will receive any message. When messages are received in the active buffer by either `pvm_recv()` or `pvm_nrecv()`, a new active buffer is created and the current receive buffer is cleared.

Whereas the `pvm_recv()`, `pvm_nrecv()`, and the `pvm_trecv()` receive their messages into a new active buffer, the `pvm_precv()` routine receives its message directly into a user-defined buffer. The taskid parameter contains the task identifier for the sending task. The messageid parameter identifies which message is being received. The buffer parameter will contain the actual message. So instead of getting the message from the active buffer using one of the unpack routines, the message is retrieved directly from the buffer parameter. The size parameter contains the length in bytes of the message. The type parameter specifies the data type of the message. The values for data type are:

PVM_STR	PVM_BYTE
PVM_SHORT	PVM_INT
PVM_FLOAT	PVM_DOUBLE
PVM_LONG	PVM_USHORT
PVM_CPLX	PVM_DCPLX
PVM_UINT	PVM_ULONG

The `pvm_trecv()` is an important routine that allows the programmer to use a timed receive. The `pvm_trecv()` routine causes the calling task to block and wait for the message, but only for the amount of time specified for the timeout parameter. The specified parameter is a structure of type `timeval` defined in `time.h`. For example:



```

#include "pvm3.h"

//...

struct timeval Timeout;
Timeout.tv_sec = 1000;
int TaskId;
int MessageId;
TaskId = pvm_parent();
MessageId = 2;
pvm_trecv(TaskId,MessageId,&Timeout);
//...

```

the Timeout variable has the tv\_sec member set to 1000 seconds. The timeval struct can be used to set the timeout values in seconds and microseconds. The timeval is a struct has the structure:

```

struct timeval{
    long tv_sec; // seconds
    long tv_usec; // microseconds
};

```

This means the pvm\_trecv() routine will block the calling task for at the most 1000 seconds. If this message gets there before the 1000 seconds have expired, the routine will return. This routine can be used to help prevent indefinite postponement and deadlock. If pvm\_trecv() is successful, it will return the number of the new active buffer. If an error occurs, then a value < 0 will be returned. If taskid = -1, the routine will accept a message from any sender. If the messageid parameter = -1, it will accept any message.

The pvm\_probe() routine determines whether a particular message has arrived from the specified sender. The taskid parameter identifies the sender. The messageid parameter identifies the particular message. If the pvm\_probe() routine sees the specified message, then the routine returns the buffer number for the new active buffer. If the specified message has not arrived, the routine returns a 0. If an error condition has occurred, the routine will return a value < 0.

## Synopsis

```

#include "pvm3.h"

int pvm_getsbuf(void);
int pvm_getrbuf(void);
int pvm_setsbuf(int bufferid);
int pvm_setrbuf(int bufferid);
int pvm_mkbuf(int Code);
int pvm_freebuf(int bufferid);

```

There are six useful buffer management routines that can be used for setting, identifying, and dynamically creating the send and receive buffers. The pvm\_getsbuf() routine is used to get the number for the active send buffer. If there is no current buffer, this routine will return 0. The pvm\_getrbuf() routine is used to get the id number for the active receive buffer. Keep in mind that every time a message is received, a new active receive buffer is created and the current receive buffer is cleared. If there is no current receive buffer, pvm\_getrbuf() will return 0. The pvm\_setsbuf() routine sets the active send buffer to bufferid. Typically, a PVM task has only one send buffer. However, sometimes multiple send buffers are required. Although only one send buffer can be active at a time, a PVM task may create additional send buffers using the pvm\_mkbuf() routine. The pvm\_setsbuf() can be used to set the

active buffer to send buffers that have been created at runtime. This routine returns the buffer identifier for the previous active send buffer. The `pvm_setrbuf()` sets the active receive buffer to `bufferid`. Remember that the PVM unpack routines work with the active receive buffer. If there is more than one buffer, then the `pvm_setrbuf()` can be used to set the current buffer to be used by the unpack routines. If the call to `pvm_setrbuf()` is successful, it will return the buffer id of the previous buffer. If the buffer identifier passed to `pvm_setrbuf()` is not valid or does not exist, then the routine can return one of the following error messages: `PvmBadParam` or `PvmNoSuchbuf`. The `pvm_mkbuf()` routine is used to create a new message buffer. The `Code` parameter specifies whether the buffer will be set up to contain data encoded as XDR format, native machine format, or pointers and sizes. The `Code` parameter can be one of three values:

<code>PvmDataDefault</code>	XDR
<code>PvmDataRaw</code>	Machine dependent (no encoding)
<code>PvmDataInPlace</code>	Pointers to the data and sizes of data only used

If the `pvm_mkbuf()` routine is successful, it will return the buffer id of the new active buffer. If an error occurs, the function will return a value  $< 0$ . For every call to `pvm_mkbuf()` there should be a call made to `pvm_freebuf()` when the send buffer is no longer needed. The memory allocated by the `pvm_mkbuf()` routine is released by `pvm_freebuf()`. `pvm_freebuf()` should only be used on a buffer that is no longer needed, for example, after the message has been sent.

## 6.4 Accessing Standard Input (stdin) and Standard Output (stdout) within PVM Tasks

A PVM environment ties a collection of machines together and presents them to the program as one logical machine with multiple processors. Which machine in the PVM should act as the console? When a PVM task inserts data into the `cout` ostream object, where will the data be displayed? If a PVM task attempts to get data from a keyboard, which keyboard will it read the data from? The `stdout` for each child process is intercepted and sent to a designated PVM task as a PVM message. Each child process inherits information that determines which task will receive information written to `stdout` and how that information should be identified. Each child process's `stdin` is tied to `/dev/null`. Anything written to `/dev/null` disappears. If `/dev/null` is opened for reading, the equivalent of end-of-file is returned. This means child processes should not be designed to rely on input from `stdin` (`cin`) or on sending output to `stdout` (`cout`). When designing input and output processing, this behavior of `stdin` and `stdout` in a PVM environment must be considered. However, `stdin` and `stdout` for the main or parent task behaves as expected. PVM tasks use messages to communicate. Input may be retrieved from messages, pipes, shared memory, environment variables, command-line arguments, or files. Output may be written to messages, pipes, shared memory, and files.

### 6.4.1 Retrieving Standard Output (cout) from a Child Task

Output written to `stdout` or inserted into `cout` behaves differently for PVM-spawned children. The parent decides what ultimately happens to the output. When output from a spawned child is inserted into `cout` or `cerr`, it is intercepted by the `pvm` for that task and is packaged into standard PVM messages and sent to a `TaskId` specified by the parent. The parent associates a pair (`TaskId`, `Code`) to the `cout` and `cerr` of its children. This is done using the `pvm_setopt()` routine. This routine is called before the children are spawned. If the `TaskId` is 0, the messages will go to the master `pvm`, where

they will be written to its error log. A spawned child may only set the TaskId to 0, the value inherited from its parent, or its own TaskId. This means the parent ultimately controls where cout or cerr would write to. A child PVM designate other PVM tasks to receive data inserted into cout or cerr. The typical approach is to let the spawning task manage any important data written to stdout or stdin and let the master pvmd take everything else.

## Summary

The PVM library is a flexible library that supports the major models of parallel programming. The advantage of a PVM environment is its ability to work with heterogeneous collections of computers that may consist of different processor speeds, sizes, and architectures. Besides hardware compatibility, it works nicely with the C++ standard library and with the UNIX/Linux system library. When combined with the C++ template capabilities, object-oriented programming capabilities, and collection of algorithms, the power of the PVM environment is increased considerably. The template facility has a nice application to SPMD programming. The containers and algorithms can be used to enhance the MIMD (MPMD) capabilities of the PVM. In [Chapter 13](#), we dig a little deeper into the PVM and show how it can be used to help implement blackboards using C++. The blackboard is one of our primary choices for implementing parallel problem solving.

# Chapter 7. Error Handling, Exceptions, and Software Reliability

"It is always possible to invent over-elaborate models to explain a set of observable facts, but the scientist, if not the philosopher, will always accept the simplest theory that is consistent with all the data."

—Alastair Rae, Quantum Physics Illusion or Reality

In this Chapter

- [What is Software Reliability?](#)
- [Failures in Software Layers and Hardware Components](#)
- [Definitions of Defects Depend on Software Specifications](#)
- [Recognizing Where to Handle Defects versus Where to Handle Exceptions](#)
- [Software Reliability: A Simple Plan](#)
- [Using Map Objects in Error Handling](#)
- [Exception Handling Mechanisms in C++](#)
- [Event Diagrams, Logic Expressions, and Logic Diagrams](#)
- [Summary](#)

One of the primary goals of software design and engineering is to produce software that meets the user's requirements correctly and reliably. Users demand reliable and correct software regardless of the software's function. Gamers have high expectations for their software in the same way as users in a business environment. Unreliable software, whether in financial, industrial, medical, scientific, or military applications, can have devastating ramifications. The dependency on software by people and machines at all levels in our society requires that every effort be made to produce reliable, robust, and fault-tolerant software. This necessity presents additional challenges to the software designer and developer who has to develop systems that contain concurrency. Programs that contain concurrency or components that execute in distributed environments contain more layers of software. The more layers involved, the more complexity that must be managed. The more complexity that needs to be managed, the greater possibility that software defects will go undiscovered. The more defects a piece of software contains, the stronger the guarantee that the software will fail, doing so at the worst possible time.

Programs divided into concurrently executing or distributed tasks have the additional challenges that are found in the process of correctly identifying the WBS (Work Breakdown Structure) of a solution. Also, the problems that are inherently part of network communications have to be handled. In addition to WBS and communication problems, synchronization woes such as deadlock and data race must be tackled. Concurrent programming is almost by definition more complex than sequential programming and therefore the error handling and exception handling for concurrent programs require more thought, more effort, and more coding. The interesting thing to note here is that the trend for software development is toward applications that require parallel and distributed programming. The Internet and the intranet model are pervasive in today's software design. General-purpose computers with multiple processors are becoming the norm rather than the exception. Embedded and industrial computing devices are becoming more sophisticated with more onboard computing power and multiple processors. The notion of the cluster is quickly becoming the de facto standard for server deployment. It is our contention that today's software designer and developer have little choice but to understand how to design and develop reliable software for parallel processor or networked environments. The requirements for software are growing in complexity and sophistication.

In many of the code examples in this book we do not show the necessary error handling or exception handling code because it would detract from some idea or concept that we are presenting. It is important to keep in mind that the examples presented in this book are introductory in nature. In practice, the amount of error handling and exception handling code for programs that require concurrency or distribution is significant. Error handling and exception handling must be part of the design of the software at every phase of its development. We advocate a modeling approach toward discovering the parallelism within a problem domain or solution domain. It is during this modeling phase that the exception handling and error handling models need to be developed. In [Chapter 10](#), we discuss how the UML (Unified Modeling Language) can be used to visualize the design of systems requiring concurrency or distributed programming. The design of error handling and exception handling techniques can also take advantage of the UML and visualization process. There is no real substitute for this visualization process. Therefore, as an initial goal you want to see your software's reliability using tools like the UML, event diagrams, event expressions, synchronization diagrams, and so on. In this chapter we take advantage of several design techniques that aid with the visualization of error and exception handling design. We also take advantage of C++'s exception handling facilities, including the exception class hierarchy, to act as a foundation for developing reliable and robust software.

## 7.1 What is Software Reliability?

Software reliability is the probability of failure-free operation of a computer program for a specifiedin a specified environment. Ideally, that probability should be as close to 100% as necessary. When failure is not an option, the software must be designed using the techniques of fault-tolerant programming. A fault-tolerant system is one that corrects or survives software faults. A fault is a program defect that can cause a piece of software to fail. We define "software failure" as the execution of some component of software that deviates from system specifications. We rely on Musa, Iannino, and Okumoto in their work *Software Reliability* for a complete characterization of faults and failures:

A fault is the defect in the program that when executed under particular conditions, causes a failure. There can be different sets of conditions that cause failures, or the conditions can be repeated. Hence a fault can be the source of more than one failure. A fault is a property of the program rather than a property of its execution or behavior. It is what we are really referring to in general when we use the term "bug." A fault is created when a programmer makes an error.

The errors that a programmer or software developer makes may be from a misinterpretation of the software requirements, or from a poor, incorrect, or incomplete translation of the software requirements into code. When the programmer makes these kinds of errors, he or she introduces defects or faults into the software. When those defects or faults are executed, they can cause software failure. Software failure can only occur during the execution. The process of testing and debugging software removes faults from software, thereby preventing the possibility of software failure. Note that we use the terms "defect" and "fault" interchangeably. We use the term "error" to refer to the mistakes that the programmer makes that introduce faults (defects) into the software. Fault tolerance is a property that allows a piece of software to survive and recover from the software failures caused by faults introduced into the software as a result of human error. The most robust fault tolerance can even correct these failures.

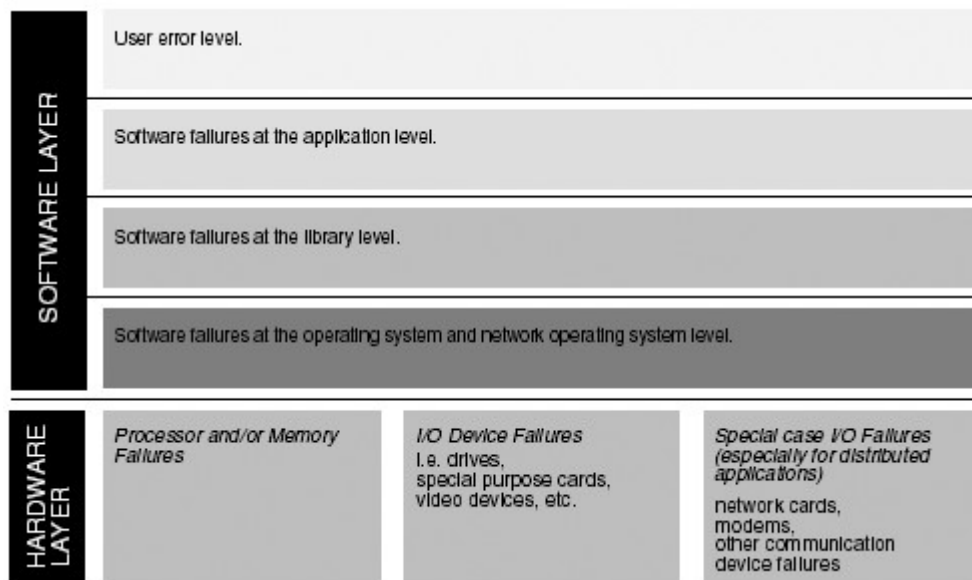
Some failures are the result of software faults. Other failures are the result of exceptional conditions (not necessarily due to human error) that can occur in either hardware or software. For instance, a network card damaged as a result of a power surge can cause the software that depends on it to fail. A virus may corrupt a data transmission that will cause the software that depends on the data transmission to fail. A user may inadvertently remove critical components of a system, thereby causing the software to fail. These kinds of failures are not due to defects in the software, but are created by conditions that

we call exceptions. An exception is an abnormal condition, exceptional circumstance, or an extraordinary occurrence that the software encounters that causes all or part of the software to fail. Although both defects and exceptions cause software failure, it is important to distinguish between them. The techniques for dealing with defects and exceptions can be and usually are different. While the end result of applying those techniques is reliable software, exception handling and error (defect) handling use different design approaches and coding constructs.

## 7.2 Failures in Software Layers and Hardware Components

Designing reliable fault-tolerant software requires that we design software that continues to operate even after some of its components fail. These components can be either hardware or software components. If our software is fault tolerant, it will have features that counter the effects of hardware or software faults. At the very minimum, our fault-tolerant designs should provide for graceful degradation of service as opposed to immediate interruption of service. If our software is fault tolerant and it encounters failed component(s), it should continue to function but at reduced levels. The failures that our software must handle can be divided into two categories: software and hardware. [Figure 7-1](#) contains a breakdown of some of the hardware components as well as the layers of software that may be involved in failure.

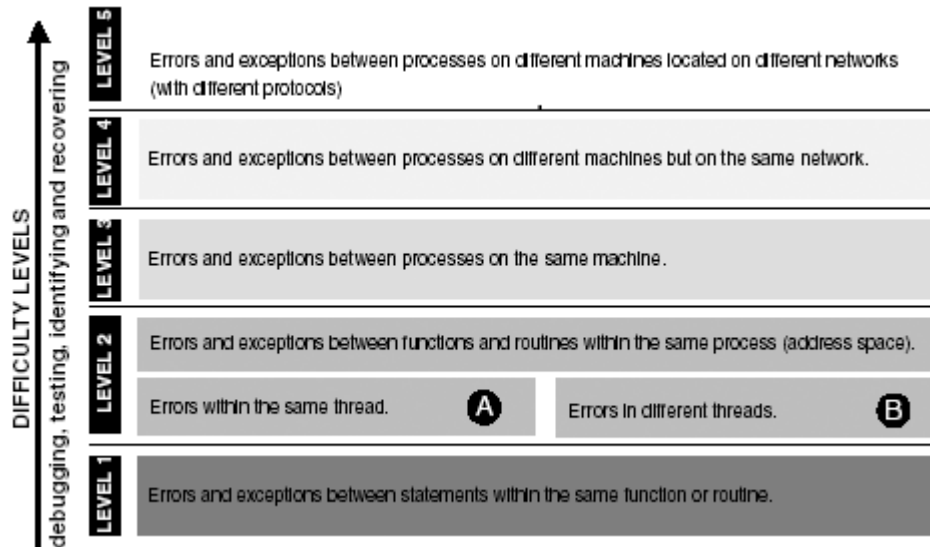
**Figure 7-1. A breakdown of some of the hardware components and layers of software involved in failure.**



In [Figure 7-1](#), we make a distinction between the hardware components and the software layers because the techniques for handling hardware failures are often different from the techniques used to handle software failures. Also in [Figure 7-1](#), there are several software layers involved. Some of the software layers are beyond the direct control of the developer and require special consideration during exception and error handling. The software design, development, and testing phases have to take into account the kinds of problems caused by hardware failures and the software layers where failure can occur. Programs that require parallelism or that consist of distributed components have additional hardware failures to consider. For instance, distributed programs rely on communications hardware and software. Failure in a communication component can cause the entire system to fail. Programs designed for parallel processors may fail if the anticipated number of processors is not available. Also, if communications or processors are available during startup, failure may occur at some time after the program has begun to execute. Exceptions may occur with any of the hardware components and in any of the software layers. In addition, each software layer may contain defects that must be handled.

During the software design phase it is useful to approach exception and error handling layer by layer. The options for recovery or repair for an application that faces failure at layer 2 are different from the options that are available at layer 3. In addition to the failures that may occur in the various software layers and hardware components, the failures may also be characterized by location. [Figure 7-2](#) depicts how as the distance between the tasks increases, so does the level of difficulty of error and exception handling.

**Figure 7-2. Contrasting the increase of distance between location of tasks and the increase of the level of difficulty of error and exception handling.**



The more distance in software or hardware components between the concurrently executing tasks, the more sophistication required when designing exception and error handling components. So from [Figures 7-1](#) and [7-2](#), we can see that in order to design and develop reliable software, we will have to make provisions for the what and where of defects and exceptions.

### 7.3 Definitions of Defects Depend on Software Specifications

A software specification is the measuring stick that we use to decide whether a piece of software has defects. We cannot determine a software component's correctness without access to the software's specification. The specification contains the description and requirements for what a software component is supposed to do and what it is not supposed to do. It is important to note that complete, thorough, and accurate specifications are notoriously difficult to produce. Specifications typically fall between two extremes: The specification may come as a set of formal documents and requirements compiled by end users, analysts, user interface engineers, domain specialists, and others, or it may only have been a set of goals and loosely defined objectives verbally communicated to the software designers and developers. The deviation of a software component from the software specification is a defect or fault. The higher the quality of the specification, the easier it is to define what a defect is or to identify where the programmer made mistakes. When a project's specification is vague, and the elements are poorly defined and the requirements are not definitive, then the definition of a software defect for that project is a moving target. If the specifications are ambiguous, we cannot say what is defective and what is not. We cannot state with certainty whether the developer is correct. Vaguely defined specifications lead to vaguely defined defects. Fault-tolerant and reliable software is not possible under these situations.

## 7.4 Recognizing Where to Handle Defects versus Where to Handle Exceptions

In general, software defects (which are the result of programmer error) should be detected and corrected during the testing phases defined in [Table 7-1](#).

**Table 7-1. Types of Testing Used During the Software Development Process**

<b>Types of Testing</b>	<b>Description</b>
Unit testing	The software is tested one component or unit at a time. A unit is described as a software module, a collection of modules, a function, a procedure, an algorithm, an object, or a program.
Integration testing	An assembly of components of the software is tested. The components are collected into logical groups and each group is tested as a unit. The groups can be subjected to the same tests. As groups pass the test, they are added to an assembly, which in turn must be tested. The number of elements that must be tested will grow combinatorially.
Regression testing	Modules are retested once they are changed. Regression testing guarantees the changes to the component do not cause it to lose any functionality.
Stress testing	Testing that pushes a component or system to and beyond its limits. This will include testing boundary conditions, which help in determining what happens at the boundaries.
Operational testing	Test the system in full operation. The software is placed in a live environment to be tested under a complete system load. This testing is also used to determine performance in a totally foreign environment.
Specification testing	The component is audited against the original specifications. The specification dictates what components are involved in the system and the relationships between those components. This is part of the software verification process.
Acceptance testing	Testing performed by the end user of the module, component, or system to determine performance. This is part of the software validation process.

Through the process of testing and debugging, defects should be identified and removed. On the other hand, exceptions are handled during execution of the program at runtime. We also distinguish between exceptional conditions and unwanted conditions. For instance, if we have designed a program that will add a list of numbers typed in by the user and the user types in some numbers and some characters that are not numbers, then this is an unwanted condition, not an exceptional condition. We should design programs to be robust through input validation so that the user is forced to enter the data that our program requires for proper execution. If part of a program that we design saves information to external storage and the program encounters an out-of-space condition, then the out-of-space condition is an



unwanted situation, not an exceptional or extraordinary condition. We reserve exception handling for the unusual, not the unwanted. We reserve exception handling techniques for the unanticipated. Situations that are unwanted but have a reasonable probability should be handled by ordinary program logic such as the following:

```
If Input data not acceptable then
    request input data again
else
    perform required operation
end if
```

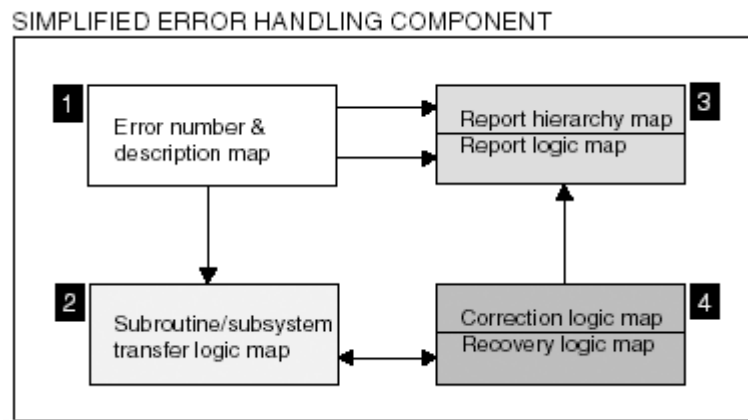
Checking conditions in this way is part of the fundamental art of programming. This kind of programming prevents problems from happening. It certainly doesn't rise to the definition of exception. There is a difference between defects and exceptions and between exceptions and unwanted conditions. Defects are dealt with using testing and debugging. Unwanted conditions are handled within the confines of the regular program logic and exceptions are handled using exception handling programming techniques. [Table 7-2](#) contrasts the difference between the characteristics of error handling, exception handling, and the handling of unwanted conditions.

**Table 7-2. Differences between the Characteristics of Error and Exception Handling and the Handling of Unwanted Conditions**

<b>Error Handling</b>	<b>Exception Handling</b>	<b>Handling Unwanted Conditions</b>
<ul style="list-style-type: none"> <li>• Logical errors discovered during design and testing</li> </ul>	<ul style="list-style-type: none"> <li>• Describes unanticipated conditions during execution time</li> </ul>	<ul style="list-style-type: none"> <li>• Describes unwanted conditions that have a reasonable probability of occurring during execution time</li> </ul>
<ul style="list-style-type: none"> <li>• Correct programs do not contain errors</li> </ul>	<ul style="list-style-type: none"> <li>• Correct programs can encounter exceptions</li> </ul>	<ul style="list-style-type: none"> <li>• Correct programs may encounter unwanted conditions</li> </ul>
<ul style="list-style-type: none"> <li>• Use program logic to anticipate and correct errors</li> </ul>	<ul style="list-style-type: none"> <li>• Use exception handling to recover from exceptions</li> </ul>	<ul style="list-style-type: none"> <li>• Use program logic to correct unwanted conditions</li> </ul>
<ul style="list-style-type: none"> <li>• Normal flow of control is maintained</li> </ul>	<ul style="list-style-type: none"> <li>• Normal flow of control is disrupted</li> </ul>	<ul style="list-style-type: none"> <li>• Attempt to maintain normal flow of control</li> </ul>

The goal is to build error handling and exception handling components that can then be integrated with the other components that make up our parallel or distributed programs. The error handling and exception handling components must have the capability of identifying and reporting what the problem is as well as recovering from or correcting the problem. The recovery and correction can involve everything from prompting the user to reenter the data to restarting a subsystem within the software. Recovery and correction efforts can involve extensive file processing, database backouts, network rerouting, processor masking, device reinitialization, and for some systems, even hardware part replacements. Error and exception handling components can take on a range of forms, from simple assertion statements to smart agents whose sole purpose it is to anticipate failures and prevent them before they happen. A significant portion of any serious piece of software will be devoted to the error and exception handling components. [Figure 7-3](#) shows the architecture for a simple error handling component.

Figure 7-3. Architecture of a simplified error handling component.



Component 1 in [Figure 7-3](#) is a simple map component that contains a list of error numbers and their descriptions. Component 2 contains a map object that maps the error numbers to jump locations, functions, or subsystems. Depending on what the error number is, component 2 is used to determine where to transfer. Component 3 is a map that maps the error numbers to the report hierarchy and report logic. The report hierarchy contains who or what should be notified of the error. The report logic determines what the notification should be. Component 4 contains two map objects. The first object maps the error numbers to objects whose purpose it is to correct some failure condition. The second object maps error numbers to objects who will return the system to a stable or at least a partially stable state. The simple error handling component in [Figure 7-3](#) can be applied to software of all sizes and shapes. How the error handling and exception handling components are used will be determined by the amount of software reliability desired.

## 7.5 Software Reliability: A Simple Plan

Keep in mind that we distinguish between error conditions and inconvenient/unwanted conditions. Inconvenient or unwanted conditions should be handled by the normal program logic. Errors (defects) require special processing. C++ Programming Language (Stroustrup, 1997) lists four basic alternative actions that a program can take when it encounters an error. According to Stroustrup, upon detecting a problem that cannot be handled locally, the program could:

- Option 1. Terminate the program.
- Option 2. Return a value representing an "error."
- Option 3. Return a legal value and leave the program in an illegal state.
- Option 4. Call a function supplied to be called in case of error.

These four alternatives are generally seen in producer-consumer relationships of all sizes. The producer is typically a piece of code that implements a library function, class, class library, or application framework. The consumer is typically a piece of code that calls a library function, class, class library, or application framework. The consumer makes a request. The producer encounters an error in attempting to fulfill the request, and the four alternatives immediately become applicable. The problem with these four alternatives is that none of them is applicable in every situation.

Obviously terminating the program every time an error occurs is simply not acceptable. We agree with Stroustrup. We can and must do better than program termination upon encountering an error. With option 2, simply returning an error value may help in some situations but not in others. Not every return value can be interpreted as success or failure. For example, if a function has a return value of floats and the range of the function includes both negative and positive values, then which value of the function can be used to represent error? This is not always possible. As far as we are concerned, option 3 is also

unacceptable. The assumption will be if the value is legal, then the operation worked. This will cause problems. Option 4 is where most of the effort should be spent whether we are discussing error or exception handling.

### 7.5.1 Plan A: The Resumption Model, Plan B: The Termination Model

Once an error or exception is encountered, there are two basic plans of attack for implementing option 4. The first plan of attack is to attempt to correct the condition or adjust for the condition that caused the failure, then resume execution at the point where the error or exception was encountered. This approach is called resumption. The second approach is to acknowledge the error or exception and perform a graceful exit of the subsystem or subroutine that caused the problem. The graceful exit is accomplished by closing the appropriate files, destroying the appropriate objects, logging the error (if possible), deallocating the appropriate memory, and handling any devices that need to be dealt with. This approach is called termination, not to be confused with the notion of just abruptly exiting the program. Both plans are valid and are useful in different situations. Before we discuss how to implement resumption or termination, let's look at the support C++ has for error handling and exception handling.

## 7.6 Using Map Objects in Error Handling

A map is a simple component that can be used as a part of any error handling or exception handling strategy. A map associates one item with another. For example, a map can be used to associate error numbers with descriptions:

```
//...
map<int,string> ErrorTable;
ErrorTable[123] = "division by zero";
ErrorTable[4556] = "no dial tone";
//...
```

Here, the number 123 is associated with "division by zero." If we write:

```
cout << ErrorTable[123] << endl;
```

Then "division by zero" will be written to cout.

In addition to mapping built-in data types, we may also map user-defined objects with built-in types. Instead of simply returning a message description for each error number, we may return an object with each error number. The object can have methods designed for error correction, error reporting, and error logging. For example, if we have a user-defined object called

defect\_response:

```
class defect_response{
protected:
    //...
    int DefectNo;
    string Explanation;

public:
    bool operator<(defect_response &X);
    virtual int doSomething(void);
    string explanation(void);
    //...
};
```

We can add defect\_response objects to the map using something like:

```
//...
map<int, defect_reponse *> ErrorTable;
defect_response * Response;
Response = new defect_response;
ErrorTable[123] = Response;
//...
```

This associates a response object with error number 123. Using polymorphism, the map object can contain pointers to any defect\_response object or any object that is descended from defect response. For instance, if we have a class:

```
class exception_response : public defect_response{
    //...
public:
    int doSomething(void)
    //...
};
```

called exception\_response that is descended from defect\_response, then we may also add pointers to type exception\_response to the ErrorTable object.

```
//...
map<int,defect_reponse *> ErrorTable;
defect_response * Response;
exception_response *Response2;
Response = new defect_response;
Response2 = new exception_response;
ErrorTable[123] = Response; // Stores an object of type
                           defect_response
ErrorTable[456] = Response2; // Stores an object of type
                             exception_response
//...
```

This shows that the ErrorTable object can map different objects with different explanations and capabilities with the appropriate error number. Therefore, the references to the doSomething() method:

```
//...
defect_response *ProblemSolver;
ProblemSovler = ErrorTable[123];
ProblemSolver->doSomething();
ProblemSovler = ErrorTable[456];
ProblemSovler->doSomething();
//...
```

will each cause the ProblemSolver object to execute a different set of instructions. Although ProblemSolver is a pointer to a defect\_response object, polymorphisms allow ProblemSolver to also point to an exception\_response object or any other object descended from defect\_response. Because the doSomething() method is declared virtual in the defect\_response class, the compiler can do dynamic binding. This will ensure that the correct doSomething() method will be called at runtime. This dynamic binding is important because each descendant of defect\_response will define its own do-Something() method. We want the doSomething() method to be called based on which descendant of defect\_response is referenced. This technique allows us to associate error numbers with objects that are specialized in handling certain error conditions. Using this technique, we can make the error handling code simpler. [Example 7.1](#) shows how the return value from some function can be used to summon the appropriate error handling object:

**Example 7.1 Using a function's return values to determine the correct ErrorHandler object to access.**

```
void importantOperation(void)
{
    //...
    Result = reliableOperation();
    if(Result != Success){
        defect_response *Solver;
        Solver = ErrorTable[Result];
        Solver->doSomething();
    }
    else{
        // continue processing
    }
    //...
}
```

Notice in [Example 7.1](#) that we do not have a series of if() statements or case statements. The map object allows us to directly access the error handling object we want by index. The doSomething() method called in [Example 7.1](#) will depend on the value of Result. Obviously this is an oversimplification of the processing. For example, [Example 7.1](#) doesn't show who's responsible for memory management of the dynamically allocated objects stored in the ErrorTable map. Also, both the reliableOperation() routine and the doSomething() function might fail. So things can be a little more complicated than what is shown in [Example 7.1](#). However, the example does illustrate how a single piece of code can handle many error situations. We can do better: [Example 7.1](#) assumes that all the errors will be addressed by objects in ErrorTable. The objects in ErrorTable are either defect\_response objects or objects descended from defect\_response. What if we have multiple families of error handling classes? [Example 7.2](#) shows how we can make the importantOperation() more general using templates.

**Example 7.2 Using a template in the importantOperation() function.**

```
template<class T,class U> int importantOperation(void)
{
    T ErrorTable;
    //...
    U *Solver;
    //...
    Solver = ErrorTable[Result];
    Solver->doSomething();
    //...
};
```

In [Example 7.2](#), ErrorTable is not restricted to defect\_response objects. This technique allows us to further simplify and expand the flexibility of our error handling code. This example uses both vertical and horizontal polymorphism. This kind of polymorphism is extremely useful in SPMD and MPMD programs. See [Chapter 9](#) for a discussion on simplifying programs that require concurrency using templates and polymorphism. Using map objects and error handling objects are important steps in the direction of increasing software reliability. We can also take advantage of the exception handling mechanism and the exception handling classes in C++. These facilities add exception handling to error handling techniques.

## 7.7 Exception Handling Mechanisms in C++

Ideally, the testing and debugging process will remove all defects from the software or at least as many defects as possible from the software. Unwanted and inconvenient conditions should be handled by regular program logic. After defects are removed and unwanted or inconvenient conditions are handled, everything left is an exception. Exception handling is supported in C++ by three keywords: try, throw, and catch. Any code that encounters an exceptional condition that it cannot cope with throws an exception hoping that some exception handler (somewhere) can handle the problem (Stroustrup, 1997). The throw keyword is used to throw an object of some type. Throwing the object transfers control to an exception handler coded to deal with the type of object thrown. The catch keyword is used to identify handlers designed to catch exception objects. For example:

```
void importantOperation
{
    // executeImportCode()
    // the Impossible Happens Somehow
    impossible_condition ImpossibleCondition;
    throw ImpossibleCondition;
    //...
}

catch (impossible_condition &E)
{
    // Do something about E
    //...
}
```

The importantOperation() routine attempts to do its work and encounters an unusual condition that it cannot cope with. In our example, it creates an object of type impossible\_condition and uses the keyword throw to throw the object. The block of code that uses the catch keyword is designed to catch objects of type impossible\_condition. This block of code is called an exception handler. Exception handlers are associated with blocks of code contained within a try expression. A try block is used to surround code that possibly contains a routine that will encounter some exceptional situation. A catch block may only follow a try block or another catch block. So we might have:

```
try{
    //...
    importantOperation()
    //...
}

catch(impossible_condition &E)
{
    // do something about E
    //...
}
```

Here, either the routine importantOperation() or one of the routine importantOperation() calls have the potential to encounter some condition that it simply cannot cope with. The routine will throw an exception. Control will be transferred to the first exception handler that accepts an error of type impossible\_condition. Either that routine will handle the exception or throw the exception to be handled by another exception handler. The objects thrown can be user-defined objects that can form simple to sophisticated error codes and messages. They may contain code that will help the exception handler perform its work. If we used objects like exception\_response objects from [Examples 7.1](#) and [7.2](#), they may be used by the exception handler to either correct the problem or allow the program to somehow

recover its state. We may also use the built-in exception classes to create exception objects.

### 7.7.1 The Exception Classes

The standard C++ class library has nine exception classes divided into two basic groups. [Table 7-3](#) shows the runtime error group and the logic error group. The runtime error group represents errors that are somewhat difficult to prevent. The logic error group represents errors that are "theoretically preventable."

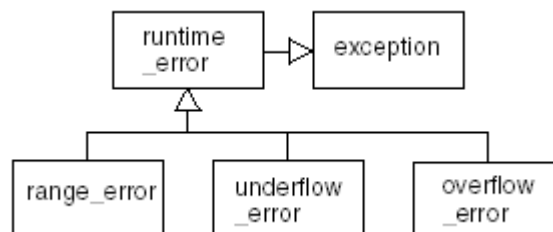
**Table 7-3. Runtime and Logic Error Classes**

<b>runtime error Classes</b>	<b>logic error Classes</b>
range_error	domain_error
underflow_error	invalid_argument
overflow_error	length_error
	out_of_range

#### 7.7.1.1 The runtime\_error Classes

[Figure 7-4](#) shows the class relationship diagram for the runtime\_error family of classes. The runtime\_error family of classes is derived from the exception class. Three classes are derived from runtime\_error: range\_error, overflow\_error, and underflow\_error. The runtime\_error classes report internal computation or arithmetic errors. The runtime\_error classes get their primary functionality from the exception class ancestor. The what() method, assignment operator=(), and the constructors for the exception handling class provide the capability of the runtime\_error classes. The runtime\_error classes provide an exception framework and architectural blueprint to build upon.

**Figure 7-4. The class relationship diagram for the runtime\_error family of classes.**



They offer little inherent functionality—the programmer must specialize them through inheritance. For example, the defect\_response and exception\_response classes created in [Examples 7.1](#) and [7.2](#) might be derived from either runtime\_error or logic\_error classes. Let's look at how the basic exception classes work with no specialization. [Example 7.3](#) shows how an exception object and a logic\_error object can be thrown.

### Example 7.3 Throwing an exception object and a logic\_error object.

```
try{
    exception X;
    throw(X);
}

catch(const exception &X)
{
    cout << X.what() << endl;
}

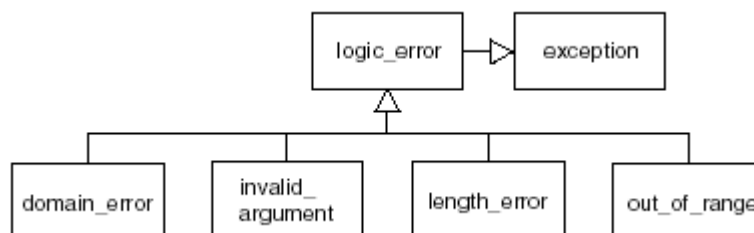
try{
    logic_error Logic("Logic Mistake");
    throw(Logic);
}
catch(const exception &X)
{
    cout << X.what() << endl;
}
```

The basic exception classes have only construction, destruction, assignment, copy, and simple reporting capabilities. They do not contain the capability to correct a fault that has occurred. The error message returned by the `what()` method of the exception classes will be determined by the string passed to the constructor for the `logic_error` object. In [Example 7.3](#), the string "Logic Mistake" passed to the constructor will be returned by the `what()` message in the catch block.

#### 7.7.1.2 The `logic_error` Classes

The `logic_error` family of classes is derived from the `exception` class. In fact, most of the functionality of the `logic_error` family of classes is also inherited from the `exception` class. The `exception` class contains the `what()` method, used to report to the user a description for the error being thrown. Each class in the `logic_error` family contains a constructor used to tailor a message specific to that class. [Figure 7-5](#) shows the class relationship diagram for the `logic_error` classes.

Figure 7-5. The class relationship diagram for the `logic_error` family of classes.



Like the `runtime_error` classes, these classes are really designed to be specialized. Unless the user adds some functionality to these classes, they cannot do anything other than report the error and the type. The nine generic exception classes provide no corrective action or error handling.



### 7.7.1.3 Deriving New Exception Classes

The exception classes can be used as-is, that is, they can be used simply to report an error message describing the error that has occurred. However, this is virtually useless as an exception handling technique. Simply knowing what the exception was doesn't do much to increase software reliability. The real value of the exception class hierarchy is the architectural road map that they provide for the designer and the developer. The exception classes provided basic error types that the developer can specialize. Many of the exceptions that occur in a runtime environment can be placed into either the `logic_error` or `runtime_error` family of classes. To demonstrate how to specialize an exception class, let's use the `runtime_error` class as an example. The `runtime_error` class is a descendant of the exception class. We can specialize the `runtime_error` class through inheritance. For instance:

```
class file_access_exception : public runtime_error{
protected:
    //...
    int ErrorNumber;
    string DetailedExplanation;
    string FileName;
    //...
public:
    virtual int takeCorrectiveAction(void)
    string detailedExplanation(void);
    //...
};
```

Here, the `file_access_exception` inherits `runtime_error` and specializes it by adding a number of data members and member functions. Specifically, the `takeCorrectiveAction()` method is added. This method can be used to help the exception handler perform its recovery and correction work. This `file_access_exception` object knows how to identify deadlock and how to break deadlock. It also has specialized logic for dealing with viruses that can damage files as well as specialized knowledge for dealing with file transfers that get unexpectedly interrupted. Each of these situations can introduce runtime exceptions. We can use our `file_access_exception` objects with the `throw`, `catch`, and `try` facilities of C++. For instance:

```
try{
    //...
    fileProcessingOperation();
    //...
}

catch(file_access_exception &E)
{
    cerr << E.what() << endl;
    cerr << E.detailedExplanation() << endl;
    E.takeCorrectiveAction();
    // Handler Take Additional Corrective Action
    //...
}
```

This technique allows you to create `ExceptionTable` map objects similar to the `ErrorTable` map objects used in [Examples 7.1.](#) and [7.2](#) Using vertical and horizontal polymorphism will also simplify exception handler processing.

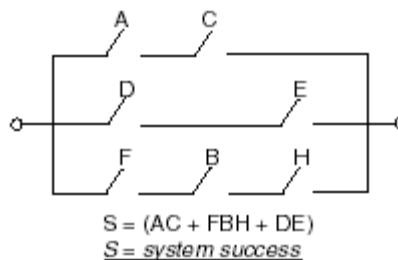
#### 7.7.1.4 Protecting the Exception Classes from Exceptions

The exception objects are thrown when some software component encounters a software or hardware anomaly. But note, the exception objects themselves do not throw exception. This has many implications. If the processing of the exception is complex enough to potentially cause another exception to be generated, then the exception processing should be redesigned and simplified where possible. The exception handling mechanism is unnecessarily complicated when exception handling code can generate exceptions. Therefore, most of the methods in the exception classes contain the empty throw() specification.

### 7.8 Event Diagrams, Logic Expressions, and Logic Diagrams

Exception handling should be used as the last line of defense because the mechanism totally alters the natural flow of control within the program. There are schemes that try to mask this fact, but those schemes are typically not flexible enough to scale to our programs that require concurrency or distribution. In the vast majority of situations where the temptation is to use catchall exception handlers, the logic can be made more robust by solid error handling or through improving the logic of a program. It is often useful to use an event diagram to help identify those components of an application that are critical to an acceptable completion of the application's work. Event diagrams can show which components can be potentially bypassed and which components lead to system failure. In some applications a single component's failure does not necessarily lead to system failure. Where a single component's failure would lead to system failure, then exception handling techniques can be used in conjunction with error handling techniques to provide the failure-is-not-an-option feature. [Figure 7-6](#) shows a simple event diagram.

Figure 7-6. A simple event diagram.



We use the event diagram to come up with a scheme to use in exception handling. [Figure 7-6](#) depicts a system that consists of seven tasks labeled A, B, C, D, E, F, and H. Notice that each label is located at a switch. If switches are closed, then the component is functioning; otherwise, the component is not functioning. The terminal point at the left represents the beginning execution and the terminal at the right represents the end of execution. In order for the program to successfully end, a path through functioning components must be found. We can illustrate how this can be applied to our exception handling situation. Lets say that we start the program executing at A. In order for the program to successfully complete, A and C must both function properly. That is, the A switch and the C switch must be closed. In this event diagram both A and C are on the same branch. This means that A and C are executing concurrently. If either A or C fails, then an exception is thrown. The exception handler could possibly start A and C again. However, our event diagram tells us that this operation will be successful if either AC or DE or FBH is successful. Therefore, we design our exception handler to execute one of an alternative set of components (e.g., DE or FBH). There is an OR relationship between AC, DE, and FBH. This means that either set of these components concurrently executing represents success. The simple event diagram in [Figure 7-6](#) indicates how we can approach our exception handler. The expression:

$$S = (AC + FBH + DE)$$

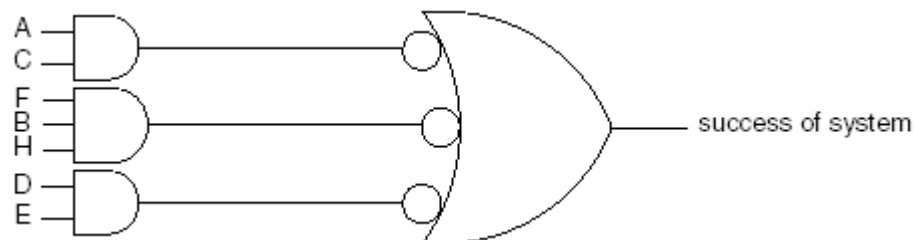
in [Figure 7-6](#) is often referred to as a logic expression or boolean expression. This expression means that (A and C) or (F and B and H) or (D and E) must successfully execute in order for the system to be in a reliable state. The event diagram can also be used to tell us which combinations of component failure can lead to system failure. For instance, if only the components E and B fail, then the system may still successfully execute if components A and C are functioning. However, if components A, H, and D were to fail, then the entire system fails. The event diagram and the logic expression are useful tools for describing concurrently dependent and independent components. They are also good for determining how to approach processing in the exception handler. For example, from [Figure 7-6](#), we can use:

```
try{
    start(task A and B)
}

catch(mysterious_condition &E){
    try{
        if(!(A && B)){
            start(F and B and H)
        }
    }
    catch(mysterious_condition &E){
        start(D and E)
    }
};
```

This kind of strategy aims at improving software reliability. Also note that the concurrency and opportunities for fault tolerant planning can be seen in the traditional logic diagram shown in [Figure 7-7](#).

**Figure 7-7. A logic diagram showing three AND gates OR'ed with OR gates to obtain the success of the system.**



[Figure 7-7](#) shows three AND gates and how they are OR'ed together to get to the S that represents the success of the system. The event diagram in [Figure 7-6](#) and the logic diagram in [Figure 7-7](#) are examples of simple techniques that can be used to visualize the critical paths and critical components in a piece of software. Once the critical paths and components are correctly identified, the developer must design software responses in case any of the critical components fail. If the termination model is used, then the exception handling does not attempt to resume execution at the point where the exception occurred; rather, the function or procedure where the exception occurred is exited, and steps are taken to put the system in as stable a state as possible. However, if the resumption model is used, the condition(s) that created the exception are either corrected or adjusted and the program resumes from the point where the exception occurred. It is important to note that the resumption model carries with it several challenges. For example, if we have a succession of nested procedure calls such as:

```

try{
    A calls B
      B calls C
        C calls D
          D calls E
            E encounters an anomaly that it cannot cope with
}
catch(exception Q)
{
}
}

```

and an anomaly occurs in E and an exception is thrown, then there is the issue of what to do about the call stack. There are also object destruction issues and suspended return values that need to be resolved. What happens if C and D are recursive? Even if we fix the condition that caused the exception in procedure E, how can we return the program to the state it was in just prior to the exception? We will have to keep stack information, object construction and destruction tables, interrupt tables, and so on. This requires a lot of overhead and cooperation between the callee and the caller. These issues represent only the surface. It is because of the complexity of implementing the resumption model and the fact that large-scale systems can be developed without it that the termination model was chosen for C++. In *The Design and Evolution of C++*, Stroustrup (1994) presents a complete rationale about why the ANSI committee eventually selected the termination model of exception handling. While the resumption model does present challenges, if the reliability and the continuity of the software are critical enough, then the effort to implement a resumption model will have to be expended and the exception handling facilities in C++ can be used to implement a resumption model.

```
// Class declaration for exception class
```

```

class exception {
public:
    exception() throw() {}
    exception(const exception&) throw() {}
    exception& operator=(const exception&) throw()
        {return *this;}
    virtual ~exception() throw() {}
    virtual const char* what() const throw();
};

```

Note the throw() declarations with empty arguments. The empty argument shows that the method cannot throw an exception. If the method attempts to throw an exception, a compile-time error message is generated. If the base class cannot throw an exception, then the corresponding method in any derived class cannot throw an exception.

## Summary

Producing reliable software is serious business. Exception handling and defect removal should be approached with extreme rigor. Thorough testing then debugging of a software component should be the primary defense against software defects. Exception handling should be added to the software system or subsystem after the software has undergone rigorous testing. Throwing exceptions should not be used as a generic error handling technique because it destroys the flow of control of the program. Exceptions should only be thrown after all of the measures have been exhausted. The standard exception handling classes should be used as architectural road maps for the programmer who wishes to design more complete and useful exception classes. If not specialized through inheritance, the standard classes can only report errors. More useful exception classes can be built that have corrective

functionality as well as more information. In general, both the termination and resumption models allow the program to continue to execute. Both models resist simply aborting the program when an error occurs. For a more complete discussion of exception handling, see *The Design and Evolution of C++* (Stroustrup, 1994).

## Chapter 8. Distributed Object-Oriented Programming in C++

"So a basic naively determined difference between the human situation and the android situation is that the human being comes equipped with an ego, whereas the robot does not."

—Cary G deBessonnet, *Towards A Sentient 'Reality' for the Android*

In this Chapter

- [Decomposition and Encapsulation of the Work](#)
- [Accessing Objects in Other Address Spaces](#)
- [The Anatomy of a Basic CORBA Consumer](#)
- [The Anatomy of a CORBA Producer](#)
- [The Basic Blueprint of a CORBA Application](#)
- [A Closer Look at Object Adapters](#)
- [Implementation and Interface Repositories](#)
- [Simple Distributed Web Services Using CORBA](#)
- [The Trading Service](#)
- [The Client/Server Paradigm](#)
- [Summary](#)

Distributed objects are objects that are part of the same application but reside in different address spaces. The address spaces may be on the same computer or on different computers connected by a network or another form of communication. The objects involved in the application could have been designed originally to work together or they may have been designed by different departments, divisions, companies, or organizations at different times and for different purposes. A distributed object-oriented application can be anything from a one-time collaborative effort by a collection of unrelated objects to a multigenerational application whose objects are spread over the entire Internet. The location of the objects can be intermixed between intranets, extranets, and the Internet. In the most general description of distributed objects, the object may be implemented in different languages such as C++, Java, Eiffel, and Smalltalk. Distributed objects play a number of roles. In some situations an object or collection of objects is used as a server that can provide database, application, or communication services. In other situations objects play the part of clients. Distributed objects can be used in collaborative problem-solving models such as blackboards and multiagent systems. Besides collaborative problem-solving models, distributed objects can be used to implement parallel programming paradigms such as SPMD and MPMD. Objects within the same application don't need any special protocol to communicate. The communication is achieved through normal method invocation, parameter passing, and global variables. Since distributed objects reside in different address spaces, inter-process communication techniques are required and in many cases network programming is necessary.

Applications that require distribution can be necessary for several reasons:

- Resources needed (e.g., databases, special processors, modems, printers, etc.) are located on different computers. Client objects interact with server objects in order to access these resources.
- Objects developed at different times, by different parties, which reside in different locations need to interoperate in order to perform some necessary work or solve some problem.
- Agents implemented as objects are highly specialized and each agent requires its own address space because it is started as a separate process.
- Objects are used as the basic unit of modularity and the modules have been implemented as separate programs, each with its own address space.
- Objects have been implemented in a SPMD or MPMD architecture in order to facilitate parallel programming, and the objects are located in different processes and on different computers.

In an object-oriented application, the work that a program does is divided between a number of objects. These objects are models of some real-world person, place, thing, or idea. The execution of an object-oriented program causes its objects to interact with each other according to the models they represent. In a distributed object-oriented application, some interacting objects will have been created by different programs possibly running on different computers. Recall from [Chapter 3](#) that each executing program has one or more processes associated with it. Each process has its own resources. For instance, each process has its own memory, file handles, stack space, process id, and so on. Tasks executing in one process do not have direct access to the resources owned by another process. If the tasks executing in one process need information stored in the memory of another process, then the two processes must explicitly exchange the information either through files, pipes, shared memory, environment variables, or sockets. Objects that reside in different processes that need to interact must also explicitly exchange information in one of these ways. The challenges for the C++ developer that wants to do distributed object-oriented programming include:

- Decomposition and encapsulation of the problem and solution into a set of objects, with the realization that some of the objects will belong to different processes and may be located on different computers.
- Communication between objects residing in different processes (address spaces).
- Synchronization of the interaction between the local and the remote objects.
- Error and exception handling in the distributed environment.

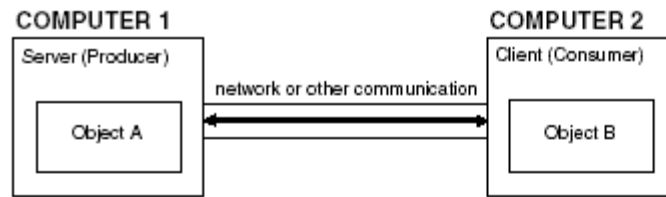
## 8.1 Decomposition and Encapsulation of the Work

Object-oriented software design is the process of translating the software requirements into a blueprint where objects model each aspect of the system to be developed and work to be done. The blueprint is centered around the structure and hierarchy of collections of objects and their relationships and interactions. The C++ class keyword is used to support the notion of a software model. There are two basic types of models. The first type of model is a scaled representation of some process, concept, or idea. This type of model is used for the sake of analysis or experimentation. For example, a class can be used to develop a molecular model. The hypothesis and structure of some chemical process within molecules can be modeled using C++'s class concept. A molecule's behavior when new groups of atoms are introduced can then be studied in software. The second type of software model is a reproduction in software of some real-world task, process, or idea. The purpose of this model is to function as its real-world counterpart as a part of some system or application. The software takes the place of some component in a manual system, or some physical thing. For example, we may use the class concept to model an adding machine. Once we have correctly modeled all of the characteristics and behavior of the adding machine, then an object can be instantiated from that class and used in place of a real adding machine. The software adding machine takes the place of the real-world adding machine. The modeled class serves as a virtual stand-in for some real-world person, place, thing, or idea. The software model captures the essence of the real thing.

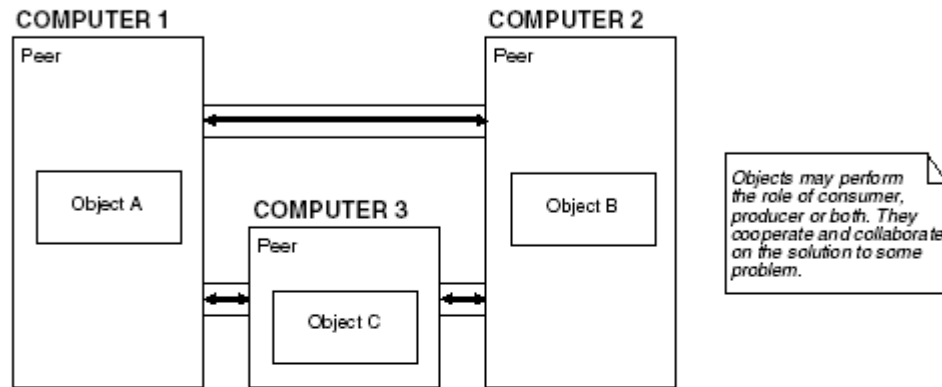
For our purposes, decomposition is the process of dividing a problem and its solution into units of work, collections of objects, and the relationships between those objects. Likewise, encapsulation is the capturing or modeling of the characteristics, attributes, and behavior of some person, place, thing, or idea using the C++ class construct. This modeling (encapsulation) and decomposition is part of the object-oriented software design phase. Object-oriented applications that contain distributed objects add an additional layer to the design considerations. In one view of the design, the locations of objects within an application should not affect the design of the attributes and characteristics of those objects. The class is a model and unless location is part of that model, the ultimate location of the objects that will instantiate that class should not matter. On the other hand, objects don't exist in a vacuum. They interact and communicate with other objects. If some of the objects that communicate are located on different computers, possibly different networks, then this consideration has to be part of the original software design process. Although there is a lot of disagreement as to where in the design process distribution needs to be considered, it must be considered. The error handling and exception handling between objects located in different processes or computers are different from the error handling and exception handling between objects that are part of the same process. Also, the communication and interaction between objects located within the same process is performed differently if those objects are located in different processes where the processes may be on different computers. This must be taken into account during the design phase. In a distributed object-oriented application, the work that must be done is divided between the objects in the application and is implemented as member functions of the various objects. The objects will be logically divided into some WBM (Work Breakdown Model). They may be divided into a client-server, producer-consumer, peer-to-peer, blackboard, or multiagent model. [Figure 8-1](#) shows the logical structure of each of these models and how the objects are distributed in each model.

**Figure 8-1. The logical structure and distribution of objects in the producer–consumer, peer-to-peer, blackboard, and multiagent models.**

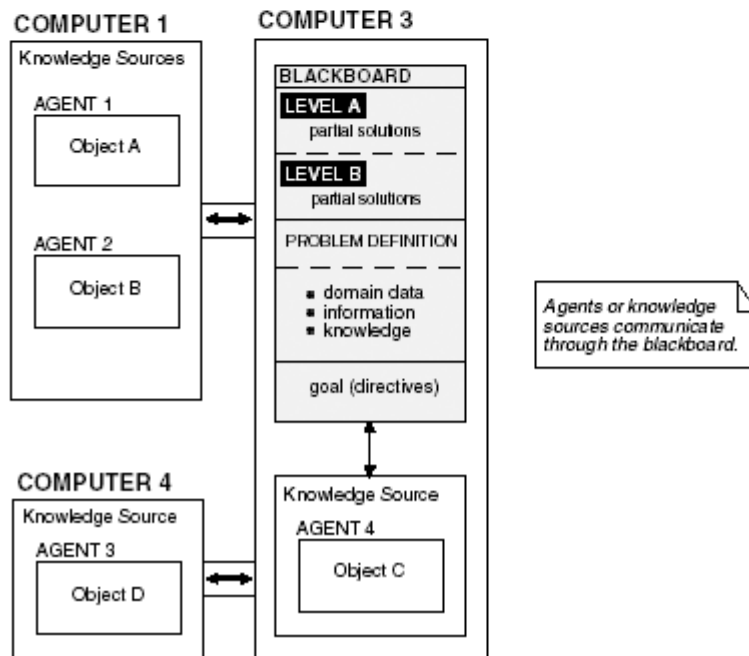
PRODUCER CONSUMER MODEL:



PEER-TO-PEER MODEL:



BLACKBOARD/MULTIAGENT MODEL:



In each model shown in [Figure 8-1](#), the objects involved may or may not be on the same computer. However, they will be in different processes. The fact that they are in different processes is what makes them distributed.<sup>[1]</sup> Each model represents a different approach to the division of work between the objects.

<sup>[1]</sup> We do not include multithreaded programs in the category of distributed programs.



### **8.1.1 Communication between Distributed Objects**

If the objects are located within the same process, then parameter passing, regular method invocation, and global variables can be used as a means of communication. If the objects are located in different processes on the same computer, then files, pipes, fifos, shared memory, clipboards, or environment variables are needed to facilitate communication between the objects. If the objects are located on different computers, then sockets, remote procedure calls, and other types of network programming will be required to facilitate communication. Not only must we be concerned with how the objects in a distributed application communicate, we must also be concerned with what they communicate. Object-oriented applications can include anything from simple to complex user-defined classes. These classes are often communicated between objects. So not only do distributed objects need to communicate simple built-in data types such as ints, floats, and doubles, they also need to communicate any type of user-defined class that might be necessary to allow some object to complete its work. Also, one object needs a way to be able to invoke methods of another object located in another address space. To complicate matters, there needs to be some way for one object to know the methods of a remote object. While C++ does support pure object-oriented programming, it does not have distributed communication facilities built in. It does not have built-in methods for locating and querying remote objects.

There are several important protocols for distributed object communication. Two of the most important protocols are IIOP (Internet Inter-ORB Protocol; pronounced "eye-op"), and the RMI (Remote Method Invocation). Using these protocols, objects located virtually anywhere on any network can communicate. In this chapter, we will discuss techniques for implementing distributed object-oriented programs using these protocols and the CORBA (Common Object Request Broker Architecture) specification. The CORBA specification is the industry standard for specifying the relationships, the interaction, and the communication between distributed objects. IIOP and GIOP are the two primary protocols that the CORBA specification works with. These protocols operate well with TCP/IP. CORBA is the easiest and most flexible way to add distributed programming to the C++ environment. The facilities provided by a CORBA implementation support the two major models of object-oriented parallelism that we use in this book: blackboards and multiagent systems. Because the CORBA specification reflects object-oriented programming, applications ranging from the small to the very large can be reasonably implemented. In this book we use MICO<sup>[2]</sup> which is an open-source implementation of the CORBA specification. The MICO implementation supports the major CORBA components and services. C++ interacts with MICO through a collection of classes and class libraries. CORBA supports distributed object-oriented modeling at every level.

<sup>[2]</sup> Any CORBA examples in this book are implemented using MICO 2.3.3 on SuSE Linux and MICO 2.3.7 on Solaris 8.

### **8.1.2 Synchronization of the Interaction between the Local and the Remote Objects**

Mutexes and semaphores can be used to help synchronize data and resource access between two or more objects located in different processes but on the same computer. This is because each process, although segregated, still has access to the computer's system memory. This system memory acts as a kind of shared memory between processes. However, multiple computers don't have any memory in common and therefore synchronization schemes must be implemented differently when the processes are distributed across different computers. Synchronizing access depending on the WBM used can require considerable communication between the distributed objects. For synchronization we will enhance the traditional methods of synchronization with CORBA's communication abilities.

### **8.1.3 Error and Exception Handling in the Distributed Environment**

Perhaps one of the most challenging areas of exception or error handling in a distributed environment is

the area of partial failure. In a distributed system, one or more components may fail while the other components operate under the assumption that everything is fine. In a local application where all of the components are within the same process, if one function or routine fails, it is not difficult for the entire application to know about it. This is not so for distributed applications. A network card might fail on one computer and the other objects executing on other computers will have no knowledge that a failure has happened. What happens if one of the objects needs to communicate or interact with an object whose network communications have been mysteriously interrupted? In a peer-to-peer model of problem solving where we have groups of objects working on various facets of some problem and one of the groups fails, how will the other groups know? Furthermore, what do we do about it? Should a single component's failure lead to system failure? If one client fails, should we shut the server down? If the server fails, should we shut the client down? What if the server or the clients only partially fail? So, in addition to data race and deadlock, we must also find ways to cope with partial failure of a distributed system where one or more of the components in the system have totally or partially failed. Again, what is necessary is a distributed approach to C++'s exception handling mechanism. The CORBA facilities provide a sufficient start.

## 8.2 Accessing Objects in Other Address Spaces

Objects that share the same scope can interact. They can access each other through their names, aliases for their names, or through pointers. An object can only be accessed where its name or a pointer to it is visible. Scope determines the visibility of object names. C++ has four basic levels of scope:

- block scope
- function scope
- file scope
- class scope

Recall that a block is defined in C++ by `{}` so that assigning `Y` to `X` in [Example 8.1](#) would be illegal because `Y` is only visible within the block that it is declared in. The function `main()` does not know the name `Y` after the closing brace of the block where `Y` was declared.

### Example 8.1 Simple example of block scope.

```
int main(int argc, char argv[])
{
    int X;
    int Z;
    {
        int Y;
        Z = Y;    // Legal
        //...
    }
    X = Y ;      // Illegal, Y is no longer defined
}
```

However, the name `Y` is visible to any other code that occurs in the same block where `Y` is declared. A name has function scope when it is declared within the function or the function's declaration. In [Example 8.1](#), `X` and `Z` are visible only to the function `main()` and cannot be accessed by other functions. File scope refers to source files. Since a C++ program can consist of multiple files, we can have objects that are visible within one file but not in another. Names that have file scope visibility are visible from the point they are declared until the end of the source file. Names with file scope visibility will not be declared in any particular function. They are usually referred to as global variables. Names that have

object scope are visible to any member function declared as part of the object. We use scope as the first level of access to an object's capabilities. The object's private, protected, and public interfaces determine the second level. Although an object's name may be visible, private and protected members still have restricted access. Scope simply tells us if the object's name is visible. In a nondistributed program, scope is associated with a single address space. Two objects in the same address space can refer to each other by name or pointer and can interact simply by invoking each other's methods.

**Example 8.2 Using objects which invoke methods of other objects of the same address space.**

```
//...
some_object  A;
another_object  B;
dynamic_object *C;
C = new dynamic_object;

//...
B.doSomething(A.doSomething());
A.doSomething(B.doSomething());
C->doMore(A.doSomething());
//...
```

In [Example 8.2](#), objects A and B are within the same scope, B is visible to A and A is visible to B. A may call B's member functions and vice versa. How is scope affected when two objects are on different machines? What happens when B is created by another program and is in a totally different address space? How will A know of B's existence? More importantly, how will A know B's name and interface? How can A call member functions that belong to B if B is part of another program? In [Example 8.2](#), objects A and B are created at compile time and object C is created at runtime. They are part of the same program, they have the same scope, and their addresses are part of the same process. In order for a process to execute an instruction, it needs to know the address of the instruction. When the program in [Example 8.2](#) is compiled, the addresses of objects A and B are stored in the executable. Therefore, the process that executes the program in [Example 8.2](#) will know where objects A and B can be found. The address for object C is assigned during runtime. The exact location of the object C is unknown until the new() function has been called. However, the pointer C does have an address within the same space as objects A and B and therefore the process will use the pointer to get to the object. We have access to each object because we have access to their addresses either directly or indirectly. The object's variable name is simply an alias for the object's address. If the object's name is within our scope then we may access it. The trick is how we associate a remote object with our local scope. If we want to access object D that is in another address space we need some way to introduce the address of the remote object to our executing process. We need some way to associate the remote object with our local scope. We need a visible name that is an alias for an address in another process that might even be on another machine. In some cases the other machine might be on another network! It would be convenient if we could simply ask for the remote object by some agreed-upon description and receive a reference for the address of the remote object. Once we had the reference, we could then interact with the remote object in our local scope. Here is where a CORBA implementation can be used to do distributed programming.

### 8.2.1 IOR Access to Remote Objects

The IOR (Interoperable Object Reference) is the standard object reference format for distributed objects. Each CORBA Object has an IOR. The IOR is a handle that uniquely identifies the object. Whereas a pointer contains a simple machine address for an object, an IOR can contain a port number, a host name, an object key, and more. In C++ we use a pointer to access dynamically created objects. The pointer tells where the object is located in memory. When an object's pointer is dereferenced, the

address is used to access the services of that object. The dereferencing process requires more effort when the object to be accessed is located in a different address space and possibly on a different computer. The pointer must contain enough information to resolve the object's exact location. If the object is located on another network, then the pointer must contain either directly or indirectly a network address, a network protocol, hostname, port address, object key, and physical address. The standard IOR acts as a kind of distributed pointer to a remote object. [Figure 8-2](#) shows a high-level breakdown of some component contained in an IOR under the IIOP protocol.

**Figure 8-2. A high-level breakdown of some component contained in an IOR under the IIOP protocol.**

Logical Components of an IOR			
HOST	PORT	OBJECT KEY	OTHER COMPONENTS
Identifies the Internet host.	Contains the TCP/IP port number where the target object is listening for requests.	A value that maps unambiguously to a particular object.	Additional information that may be used in making invocations, e.g., security.

The notion of a portable object reference is an important advancement in distributed computing. It allows local references to remote objects to appear virtually anywhere on the Internet or an intranet. This has important implications for multiagent systems where agents may need to travel between systems and throughout the Internet. The IOR standard creates some foundation for mobile objects and distributed agents. Once your program has access to an object's IOR, then an ORB (Object Request Broker) can be used to interact with the remote object through method invocation, parameter passing, return values, and so on.

### 8.2.2 ORBS (Object Request Brokers)

The ORB acts on behalf of your program. It sends messages to the remote object and returns messages from the remote object. The ORB acts as a middleman between your objects and the remote objects. The ORB takes care of all the details involved in routing a request from your program to the remote object, and routing the response from the remote object back to your program. It makes the communications between systems virtually transparent. The ORB removes the need to do socket programming between processes on different computers. Similarly, it removes the need to do pipe or fifo programming between processes on the same computer. It takes care of much of the network programming that is required for distributed programs. Furthermore, it hides the differences between operating systems, computer languages, and hardware. The local objects are not aware of what language the remote objects have been implemented in, what platform they are running on, or whether they are located on the Internet or some local intranet. The ORB uses the IOR to help facilitate communications between machines, networks, and objects. Notice in [Figure 8-2](#) that an IOR does contain information that can be used to make TCP/IP connections. We present only a high-level partial description of the IOR because the IOR is meant to be a black box for the developer. The ORB uses the IOR to locate the target object. Once the target object is located, the ORB activates it and transmits any arguments that are necessary to call the object. The ORB waits for the request to complete and returns the necessary information to the calling object or an exception if the method invocation or call fails. [Figure 8-3](#) contains a simplified overview of the steps that an ORB uses on behalf of a local object.

**Figure 8-3. The simplified overview of the steps that an ORB uses on behalf of a local object.**

SIMPLIFIED ORB METHOD INVOCATION STEPS
1) Locate the remote object.
2) Activate the module containing the target object if it is not already activated.
3) Transmit arguments to the remote object.
4) Wait for response from the invocation of the remote object's method.
5) Return information to the local object or exception if the remote method invocation failed.

The steps in [Figure 8-3](#) present a simplified overview of what the ORB does during an interaction with a remote object. These steps are almost transparent to the local object. The local object invokes one of the methods of the remote object and the ORB performs these steps on behalf of the local object. The ORB does a lot of processing with a few simple lines of code. Typically, a distributed object-oriented application requires at least two programs. Each program has one or more objects that will interact with each other across address spaces. The object interaction may be client-server, producer-consumer, or peer-to-peer in nature. Therefore, if we have two programs, one will act as the client and the other as the server, or one as the producer and the other as the consumer, or they will both be peers. [Program 8.1](#) implements a consumer that invokes a simple remote adding machine object. The program shows how a remote object may be accessed and how an ORB is initialized and used.

**Program 8.1**

```
1 using namespace std;
2 #include "adding_machine_impl.h"
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6
7
8 int main(int argc, char *argv[])
9 {
10     CORBA::ORB_var Orb = CORBA::ORB_init(argc,argv,"mico-local-orb");
11     CORBA::BOA_var Boa = Orb->BOA_init(argc,argv,"mico-local-boa");
12     ifstream In("adding_machine.objid");
13     string Ref;
14     if(!In.eof()){
15         In >> Ref;
16     }
17     In.close();
18     CORBA::Object_var Obj = Orb->string_to_object(Ref.data());
19     adding_machine_var Machine = adding_machine::_narrow(Obj);
20     Machine->add(700);
21     Machine->subtract(250);
22     cout << "Result is " << Machine->result() << endl;
23     return(0);
24 }
25
```

On line 10 the ORB is initialized. On line 15 the IOR for the adding\_machine object is read from a file. One of the nice features of the IOR is that it can be stored as a simple string and communicated to other programs. Transmitting the IOR through command line arguments, stdin, environment variables, or files are the simplest methods. An IOR can be sent using e-mail or ftp. IORs can be shared through common file systems and can be downloaded from Web pages. Once a program has an IOR for a remote object, then an ORB can be used to access the remote object. We shall cover other techniques for communicating IORs later in this chapter. But the file system technique is enough to get us started. The IOR was originally converted from an object reference to its stringified form by the remote adding machine's ORB and written to a file. On line 18 the local Orb object converts the stringified IOR back to an object reference. On line 19 the object reference is used to instantiate an adding\_machine object. The interesting thing about this adding\_machine object is that when its methods are invoked they will cause code on the remote machine to execute. The calls on line 20, 21, and 22

```
Machine->add(700);
Machine->subtract(250);
cout << "Result is " << Machine->result() << endl;
```

although made in our local scope, refer to executable code in another address space and in this case on another machine. To the developer the Machine object's location is transparent. After the object has been created on line 19 it is used like any other C++ object. Although there are very specific differences between local object invocations and remote object invocations,[\[3\]](#) the object-oriented metaphor is maintained, and from the object-oriented programming perspective remote objects look and feel like local objects. The code in [Program 8.1](#) is client code or consumer code because it uses the services of the adding\_machine object. In order for this simple adding machine application to be complete, we need the code that implements the adding\_machine object. The code in [Program 8.2](#) shows the second component to our simple adding machine application.

[3] Remote objects invocation introduces latency, security requirements, and the possibility of partial failure.

### Program 8.2

```
1  #include <iostream>
2  #include <fstream>
3  #include "adding_machine_impl.h"
4
5
6
7
8  int main(int argc, char *argv[])
9  {
10  CORBA::ORB_var Orb = CORBA::ORB_init(argc,argv,"mico-local-orb");
11  CORBA::BOA_var Boa = Orb->BOA_init(argc,argv,"mico-local-boa");
12  adding_machine_impl *AddingMachine = new adding_machine_impl;
13  CORBA::String_var Ref = Orb->object_to_string(AddingMachine);
14  ofstream Out("adding_machine.objid");
15  Out << Ref << endl;
16  Out.close();
17  Boa->impl_is_ready(CORBA::ImplementationDef::_nil());
18  Orb->run();
19  CORBA::release(AddingMachine);
20  return(0);
21 }
22
23
```

Notice on line 10 that the producer program also has to initialize an Orb object. This is an important requirement for CORBA-based programs. Each program communicates with the aid of an ORB. Initializing the ORB is one of the first things that a CORBA program must do. On line 12, the actual adding\_machine object is declared. This is the object that [Program 8.1](#) will actually communicate with. On line 13, the object reference for the actual adding\_machine object is converted to its stringified form. It's then written to a simple text file that can be easily read. Once the IOR is written to the file, the Orb object waits for a request. Each time one of its methods is called, it performs the necessary addition or subtraction to a persistent value. This value is accessed by calling the adding\_machine's result() method. [Programs 8.1](#) and [8.2](#) represent barebones CORBA programs that show the basic structure that CORBA programs will have. The code that makes the adding\_machine object distributed begins with its CORBA class declaration. Each CORBA object starts out as an IDL (Interface Definition Language) design.

### 8.2.3 Interface Definition Language (IDL): A Closer Look at CORBA Objects

The IDL is the standard object-oriented design language used to design classes that will be used for distributed programming. It is used to express class interfaces and class relationships. It is used to specify member function prototypes, parameter types, and return types. One primary function of the IDL is to separate the class interface from the implementation. Therefore, the actual definitions of methods are not specified with the IDL. Neither the implementation of member functions nor data members are specified using IDL. The IDL only specifies the function interface. [Table 8-1](#) contains the commonly used keywords in the IDL.

**Table 8-1. IDL Keywords**

#### IDL Keywords

abstract	enum	native	struct
any	factory	Object	supports
attribute	FALSE	octet	typedef
boolean	fixed	oneway	unsigned
case	float	out	union
char	in	raises	ValueBase
const	inout	readonly	valuetype
cell	interface	sequence	void
double	long	short	wchar

## IDL Keywords

exception

module

string

The keywords in [Table 8-1](#) are reserved words in a CORBA program. In addition to specifying the function interface for a class, the IDL is used to specify relationships between classes. The IDL supports:

- user-defined types
- user-defined sequences
- array types
- recursive types
- exception semantics
- modules (similar to namespaces)
- single and multiple inheritance
- bitwise and arithmetic operators

Here is the IDL definition for adding\_machine class from [Example 8.2](#):

```
interface adding_machine{
    void add(in unsigned long X);
    void subtract(in unsigned long X);
    long result();
};
```

It begins with the CORBA keyword interface. Notice that this adding\_machine declaration does not include any variables to hold the result of the additions and subtractions. Its add() and subtract() methods accept a single unsigned long as a parameter. The parameter is accompanied by the CORBA keyword in to denote that the parameter is an input parameter. This class declaration is stored in a separate source file and named adding\_machine.idl. Source files containing IDL definitions must end in the .idl suffix. The source file containing the IDL declaration must be converted to C++ before it can be used. This conversion can be done using a preprocessor step or by a standalone program. All CORBA implementations include an IDL compiler. There are IDL compilers for C, Smalltalk, C++, Java, and so on. The IDL compiler converts IDL definitions into the appropriate language. In our case the IDL compiler converts the interface declaration into legitimate C++ code. Depending on the implementation of CORBA that you use, the IDL compiler is called with syntax that will be similar to:

```
idl adding_machine.idl
```

This command will produce a file that contains C++ code. Since our IDL definition is saved in a file named adding\_machine.idl, the MICO IDL compiler produces a file named adding\_machine.h that contains several C++ skeleton classes and some CORBA data types. [Table 8-2](#) contains the basic IDL data types.



**Table 8-2. Basic IDL Data Types**

<b>IDL Datatypes</b>	<b>Range</b>	<b>Size</b>
long	$-2^{31}$ to $2^{31} - 1$	$\geq 32$ bits
short	$-2^{15}$ to $2^{15} - 1$	$\geq 16$ bits
unsigned long	0 to $2^{32} - 1$	$\geq 32$ bits
unsigned short	0 to $2^{16} - 1$	$\geq 16$ bits
float	IEEE single-precision	$\geq 32$ bits
double	IEEE double-precision	$\geq 64$ bits
char	ISO Latin-1	$\geq 8$ bits
string	ISO Latin-1, except ASCII NULL	Variable length
boolean	TRUE or FALSE	Unspecified
octet	0-255	$\geq 8$ bits
any	Runtime identifiable arbitrary type	Variable length

Even after the IDL compiler creates C++ code from the interface class, the implementation for the interface class methods are still undefined. The IDL compiler produces several C++ skeletons that are to be used as base classes. [Example 8.3](#) shows two of several classes generated by our MICO IDL compiler from the file `adding_machine.idl`.

**Example 8.3 Two classes generated by MICO IDL compiler from the `adding_machine.idl`.**

```
class adding_machine : virtual public CORBA::Object{
public:
    virtual ~adding_machine();

    #ifdef HAVE_TYPEDEF_OVERLOAD
    typedef adding_machine_ptr _ptr_type;
    typedef adding_machine_var _var_type;
    #endif
    static adding_machine_ptr _narrow( CORBA::Object_ptr obj );
    static adding_machine_ptr _narrow( CORBA::AbstractBase_ptr obj );
    static adding_machine_ptr _duplicate( adding_machine_ptr obj );
```

```

    {
        CORBA::Object::_duplicate (_obj);
        return _obj;
    }

static adding_machine_ptr _nil()
{
    return 0;
}

virtual void *_narrow_helper( const char *repoint );
static vector<CORBA::Narrow_proto> *_narrow_helpers;
static bool _narrow_helper2( CORBA::Object_ptr obj );
virtual void add( CORBA::ULong X ) = 0;
virtual void subtract( CORBA::ULong X ) = 0;
virtual CORBA::Long result() = 0;

protected:
    adding_machine() {};
private:
    adding_machine( const adding_machine& );
    void operator=( const adding_machine& );
};

class adding_machine_stub : virtual public adding_machine{
public:
    virtual ~adding_machine_stub();
    void add( CORBA::ULong X );
    void subtract( CORBA::ULong X );
    CORBA::Long result();

private:
    void operator=( const adding_machine_stub& );
};

```

adding\_machine.idl is input to the compiler and adding\_machine.h along with its skeleton classes is output from the compiler. The developer uses inheritance to actually provide implementations for the function interfaces declared in the IDL source file. For instance, [Example 8.4](#) shows the user-defined class that provides the implementation for one of the skeleton classes produced by the IDL compiler.

**Example 8.4 User-defined class implementation of skeleton classes.**

```

class adding_machine_impl : virtual public adding_machine_skel{
private:
    CORBA::Long Result;
public:
    adding_machine_impl(void)
    {
        Result = 0;
    };
    void add(CORBA::ULong X)
    {
        Result = Result + X;
    };
    void subtract(CORBA::ULong X)
    {
        Result = Result - X;
    };
};

```

```

CORBA::Long result(void)
{
    return(Result);
};
};

```

One of the skeletons that IDL compiler creates from the `adding_machine` interface class is named `adding_machine_skel`. Notice that the IDL uses the name used in the interface definition to derive new classes. Our `adding_machine_impl` class provides the implementation for the function interfaces declared using the IDL. First, the `adding_machine_impl` class declares a data member named `Result`. Second, it declares the actual implementations for the `add()`, `subtract()`, and the `result()` methods. So while the `adding_machine` interface class specifies the declaration of these methods, the `adding_machine_impl` class provides implementation of the methods. The userdefined `adding_machine_impl` class will inherit a lot of functionality useful for distributed programming from the base class. This is the basic scheme when doing CORBA programming. An interface class is designed that represents the interfaces to be used. The IDL compiler is called to generate real C++ class skeletons from the interface definition. The developer derives a class from one of the skeletons and provides implementations for the methods defined in the interface class and data members that will be used to hold attributes of the object. Generating real C++ classes from IDL is a three-step process:

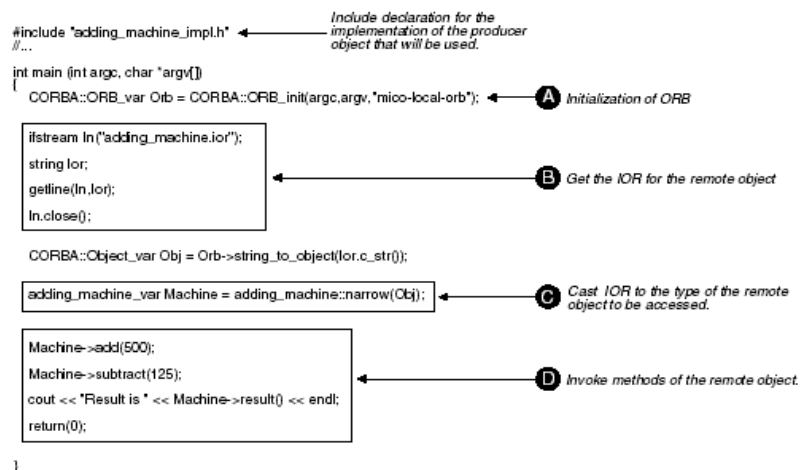
1. Design the class interfaces, relationships, and hierarchies using the IDL.
2. Use the IDL Compiler to generate real C++ skeletons from the IDL classes.
3. Use inheritance to create descendants from one or more of the skeleton classes and implement the interface methods inherited from the skeleton classes.

We'll discuss this process in more detail later in this chapter. First, let's take a closer look at the basic structure of a consumer program.

### 8.3 The Anatomy of a Basic CORBA Consumer

One of the most common models for distributed programming is the consumer-producer model. In this model, one program plays the role of producer and another plays the role of consumer. The producer creates some service or data used by a consumer. For example, we could have a program that generates unique license plate numbers upon demand. The consumer is the program that makes requests for new license plate numbers and the producer is the program that generates the license plate numbers. Typically, the consumer and producer are located in different address spaces. [Figure 8-4](#) shows several components and steps that most CORBA consumer programs contain.

Figure 8-4. Components and steps used by a CORBA consumer program.



To communicate with objects on other computers or in different address spaces, each program involved in the communication must declare an ORB object. Once the Orb object is declared, then the consumer program has access to its member functions. In [Figure 8-4](#), the ORB is initialized using the call:

```
CORBA::ORB_var Orb = CORBA::ORB_init(argc,argv,"mico-local-orb");
```

This initializes an ORB object. The CORBA::ORB\_var type is a handle to an object of type ORB. In CORBA implementations, objects that have the \_var designation take care of deallocating its underlying reference. This is in contrast to the objects that have the \_ptr designation. The command-line arguments are passed to the ORB's constructor along with an orb\_id. In our case, the orb\_id is "mico-local-orb". The string passed to the ORB\_init() function that names the ORB to be initialized is implementation specific and can differ between implementations. The derived object is referred to as the servant object.

Once the ORB and the object adapter are initialized, the next basic component that any CORBA application will need is the IOR for the remote object(s). In [Figure 8-4](#), the IOR is retrieved from a file named adding\_machine.ior. The IOR has been written in its stringified form to the file. The ORB object is used to convert the IOR from a string back to its object form using its string\_to\_object() method. In [Figure 8-4](#), this is accomplished by the call:

```
CORBA::Object_var Obj = Orb->string_to_object(Ior.c_str());
```

Here, Ior.c\_str() returns the stringified IOR and Obj will be a reference to the object form of the IOR. The object form of the IOR is then narrowed. This narrowing process is analogous to C++ type casting. The narrowing process sizes an object reference to the appropriate object type. In this case, the appropriate type is adding\_machine. The consumer program in [Figure 8-4](#) narrows the IOR object using the call:

```
adding_machine_var Machine = adding_machine::_narrow(Obj);
```

This process creates a reference to an adding\_machine object. The consumer program can now call the methods defined in the IDL interface for the adding\_machine class. For instance:

```
Machine->add(500);  
Machine->subtract(125);
```

call the add() and subtract() methods of the remote object. Although the consumer program in [Figure 8-4](#) is an oversimplified consumer, it does show the basic components of a typical CORBA consumer or client program. The consumer program requires a producer program in order for the application to be complete. We will look at a simplified CORBA program that acts as the producer for the program in [Figure 8-4](#).

## 8.4 The Anatomy of a CORBA Producer

The producer is responsible for providing either data, routines, or services to its consumer programs. The producer, together with the consumer, make a complete distributed application. Each CORBA producer program is designed with the assumption that there will be consumer programs to invoke its services. Therefore, each producer program will create servant objects and provide IORs so that the objects may be accessed. [Figure 8-5](#) contains a simple producer program used in conjunction with the consumer program from [Figure 8-4](#). [Figure 8-5](#) contains the basic components that any CORBA producer program will contain.

**Figure 8-5. Basic components of a CORBA producer program.**

```
#include <iostream>
#include <fstream>
#include "adding_machine_impl.h" ← Include declaration for the
                                implementation of the
                                adding_machine object

int main (int argc, char *argv[])
{
    CORBA::ORB_var Orb = CORBA::ORB_init(argc,argv,"mico-local-orb"); ← A Initialization of the ORB object
    CORBA::BOA_var Boa = Orb->BOA_init(argc,argv,"mico-local-boa"); ← and object adapter.

    adding_machine_impl *AddingMachine = new adding_machine_impl; ← B Instantiate the implementation
                                                                    object.

    CORBA::String_var Ior = Orb->object_to_string(AddingMachine); ← C Stringify the IOR (object reference)
    ofstream Out("adding_machine.ior"); ← and write it to a file.
    Out << Ior;
    Out.close();

    Boa->impl_is_ready(CORBA::ImplementationDef::_nil());

    Orb->run(); ← D Listen for request from
                                                                    consumer objects.

    CORBA::release(AddingMachine);
    return(0);
}
```

Notice that part A for the consumer program and the producer program are essentially the same. Both the consumer and the producer program require an ORB to communicate. The ORB is used to get a reference to an object adapter. [Figure 8-5](#) contains the call:

```
CORBA::BOA_var Boa = Orb->BOA_init(argc,argv,"mico-local-boa");
```

This call is used to get a reference to an object adapter. The object adapter is a middleman between the ORB and the object that implements the services to be called. Keep in mind that CORBA objects start as interface declarations only. At some point in the development process, a derived class provides the implementation for the CORBA interface. The object adapter acts as the middleman between the interface that the ORB interacts with and the real methods implemented by the derived class. The object adapters are used to access servant and implementation objects. The producer in [Figure 8-5](#) creates an implementation object in part B using:

```
adding_machine_impl *AddingMachine = new adding_machine_impl;
```

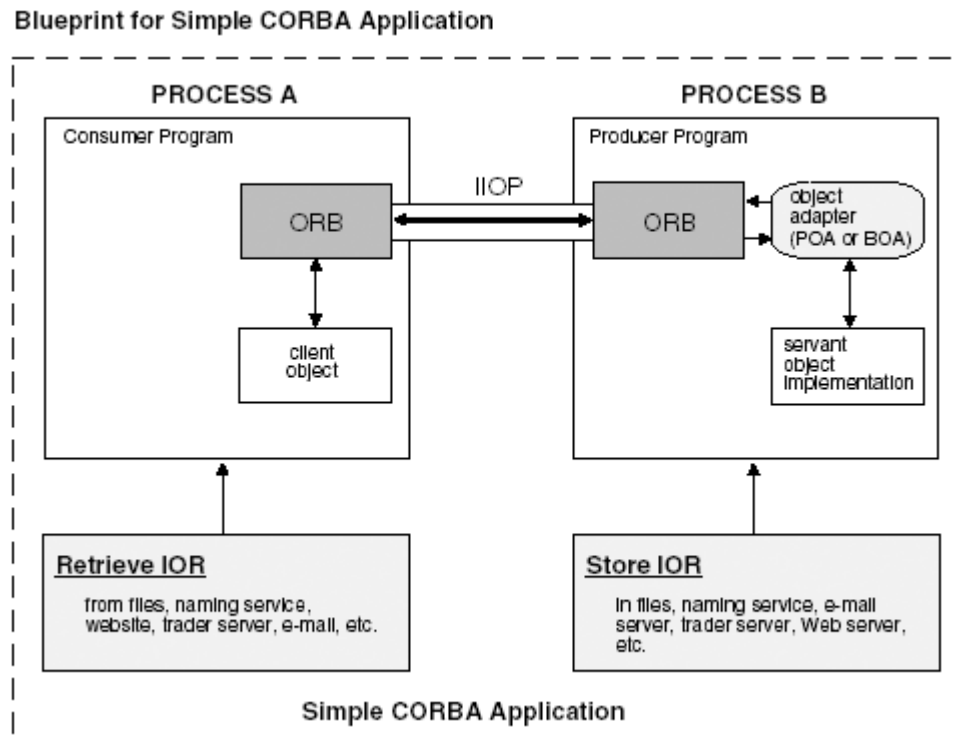
This is the object that will provide the implementation for the services that the client or consumer objects will request. Also notice that in part C in [Figure 8-5](#), the producer program uses the Orb object to convert the IOR to a string and writes the string to a file named `adding_machine.ior`. This file can be transmitted to the producer through ftp, e-mail, over http using Web pages, via NFS mounts and so on. There are other ways to communicate the IOR, but the file method provides a simple introduction. After

the IOR is written the producer program simply waits for requests from client or consumer programs. The producer program in [Figure 8-5](#) is also an oversimplification of the CORBA producer or server program, but it does contain the basic components that a typical producer program will have.

## 8.5 The Basic Blueprint of a CORBA Application

We can see from the programs in [Figures 8-4](#) and [Figures 8-5](#) that a barebones CORBA application will require two ORBs, an object adapter, a method for communicating an IOR, and at least one servant object. [Figure 8-6](#) shows the logical structure of a barebones CORBA application.

**Figure 8-6. The logical structure of a barebones CORBA application.**



After the IOR is obtained and narrowed, the remote method invocation in the consumer or client program looks just like regular method calls in a C++ program. In the CORBA examples in this book, the IIOP (Internet Inter ORB Protocol) is assumed. Therefore, the ORBs in [Figure 8-6](#) are communicating using a TCP/IP protocol. The IOR will contain enough information about the remote object's location to facilitate the TCP/IP communication. The object adapter in [Figure 8-6](#) will typically be a portable object adapter. However, some older or simpler programs may use the basic object adapter. We will describe the difference between these two adapters later in this chapter. Each CORBA application has one or more servant objects that implements the interface designed in the IDL class. The simple consumer and producer programs shown in [Figures 8-4](#) and [8-5](#) can execute on the same computer in different processes or on different computers. If the programs are executed on the same computer then the file `adding_machine.ior` should be accessible from both programs. If the programs are executed on different computers, then the file will have to be sent to the client computer via ftp, e-mail, http, and so on. The compilation and execution details for the programs shown in [Figure 8-4](#) and [8-5](#) are shown in [Profile 8.1](#) and [Profile 8.2](#)

## Program Profile 8.1

### Program Name

adding\_machine\_client\_impl.cc

### Description

This program is a simple consumer program. It connects to the CORBA producer program shown in [Figure 8-5](#). It adds 500 to the adding machine and then subtracts 125. It sends the result of the operations to cout using the results() method.

### Libraries Required

mico2.3.3 or mico2.3.7

### Headers Required

None

### Compile & Link Instructions

```
idl -poa adding_machine.idl
mico-c++ -g -c adding_machine.cc -o adding_machine.o
mico-c++ -g -c adding_machine_impl.cc -o adding_machine_impl.o
mico-c++ -g -c adding_machine_client_impl.cc -o
adding_machine_client_impl.o
mico-ld -g -o adding_machine_client adding_machine_client_impl.o
adding_machine_impl.o adding_machine.o -lmico2.3.3
```

### Test Environment

SuSE Linux 7.1 gnu C++ 2.95.2, Solaris 8 Workshop 7, MICO 2.3.3, MICO 2.3.7

### Execution Instructions

Execute the binary named adding\_machine\_client (e.g., ./adding\_machine\_client). The CORBA producer program needs to be started first. The producer program is shown in [Figure 8-5](#) and is named adding\_machine\_server.

### Notes

The CORBA producer program should be running at the time adding\_machine\_client is invoked.

## Program Profile 8.2

### Program Name

adding\_machine\_server\_impl.cc

### Description

This program is a simple server program shown in [Figure 8-5](#). It accepts requests for additions and subtractions and produces the results of those requests.

### Libraries Required

mico2.3.3 or mico2.3.7

### Headers Required

None

### Compile and Link Instructions

```
idl -poa adding_machine.idl
mico-c++ -g -c adding_machine.cc -o adding_machine.o
mico-c++ -g -c adding_machine_impl.cc -o adding_machine_impl.o
mico-c++ -g -c adding_machine_server_impl.cc -o
adding_machine_server_impl.o
mico-ld -g -o adding_machine_server adding_machine_server_impl.o
adding_machine_impl.o adding_machine.o -lmico2.3.3
```

### Test Environment

SuSE Linux 7.1 gnu C++ 2.95.2, Solaris 8 Workshop 7, MICO 2.3.3, MICO 2.3.7

### Execution Instructions

Execute the binary named adding\_machine\_server (e.g., ./adding\_machine\_server)

### Notes

None

### 8.5.1 The IDL Compiler

The IDL compiler is a tool used to translate IDL Class definitions into C++ code. This code consists of a collection of class skeletons, enumerated types, and template classes. The IDL compiler used for the CORBA programs in this book is the MICO IDL compiler. [Table 8-3](#) contains some commonly used command-line options to the IDL compiler.



**Table 8-3. Some Commonly Used Command-Line Options to the IDL Compiler**

<b>IDL Compiler Command-Line Options</b>	<b>Description</b>
--	--------------------

<code>--boa</code>	Generates skeletons that use the basic object adapter (BOA). This is the default.
<code>--no-boa</code>	Turns off code generation of skeletons for the BOA.
<code>--poa</code>	Generates skeletons that use the portable object adapter (POA).
<code>--no-poa</code>	Turns off code generation of skeletons for the POA. This is currently the default.
<code>--gen-included-defs</code>	Generate code that was included using the <code>#include</code> .
<code>--version</code>	Prints the version of MICO.
<code>-D&lt;define&gt;</code>	Defines a preprocessor macro. This option is equivalent to the <code>-D</code> switch of most UNIX C-compilers.
<code>-I&lt;path&gt;</code>	Defines a search path for <code>#include</code> directives. This option is equivalent to the <code>-I</code> switch of most UNIX C-compilers.

The `-boa` and `-poa` switches in [Table 8-3](#) can be used to determine what kind of adapter skeletons will be produced. For example, typing the command:

```
idl -poa -no-boa adding_machine.idl
```

will produce a file named `adding_machine.h` that contains skeletons for the `poa` (portable object adapter) and it will turn off the production of skeletons for the `boa` (basic object adapter). Typing the command:

```
idl -h
```

generates a complete list of the IDL compiler switches. If the man pages for the MICO distribution have been properly installed, then typing the command:

```
man idl
```

will provide a complete explanation of the IDL switches available. Designing the IDL classes is the first step in CORBA programming. The next major step in a CORBA program is determining how the IORs for remote objects will be stored and retrieved.

### **8.5.2 Obtaining IOR for Remote Objects**

The ORB class has two member functions that can be used for converting IOR objects between strings

and `Object_ptr`. The methods are `string_to_object()` and `object_to_string()`. The `string_to_object()` member function takes a `const char *` and converts it to an `Object_ptr`. The `object_to_string()` member function takes an `Object_ptr` and converts it to a `char *`. These methods are part of the ORB class interface. The `object_to_string()` method is used to stringify object IORs. Once the IOR has been stringified it can be transmitted to client and consumer programs through a variety of techniques, including:

E-mail	Shared file systems (NFS mounts)
ftp	Embedded within html documents
Java applets/servlets	Command-line arguments
Shared memory	Traditional IPC (i.e., pipes, fifos)
Environment variables	CGI get and post commands

The receiving program then takes the stringified IOR and uses its ORB's `string_to_object()` member function to convert the IOR to a CORBA object ptr. The CORBA object ptr is then narrowed and used to initialize the local object. [Programs 8.1](#) and [8.2](#) use stringified objects and a file to communicate the IOR between the consumer program and the producer program. The stringified IOR can be used to facilitate very flexible connections to remote objects that can reside virtually anywhere on the Internet or on any intranet or extranet. In fact, the MIWCO (Wireless Mico) is an open-source implementation of wCORBA,<sup>[4]</sup> the wireless CORBA standard, and can be used to enhance the mobility of objects. The wireless specification enables mobility through a MIOR (Mobile IOR). The wireless specification has support for TCP, UDP, and WAP WDP (Wireless Application Protocol Wireless Datagram Protocol) transports. Multiagent and distributed agent systems can also benefit by taking advantage of the IOR standards. The IOR and MIOR are part of the building blocks for the next generation of object-oriented Web services. It is important to note that although the stringified IOR provides a flexible and portable object reference, it may not be ideal for all situations and configurations. Moving a file containing the IOR may not be practical for many installations. Forcing client and server applications to share the same file system or network may not be practical. Security concerns might exclude the stringified IOR as an option. If a client-server application is large and diverse enough, then the stringified IOR sharing may be too restricting. The CORBA specification includes two other standards for obtaining or communicating object references: naming services, and trading services.

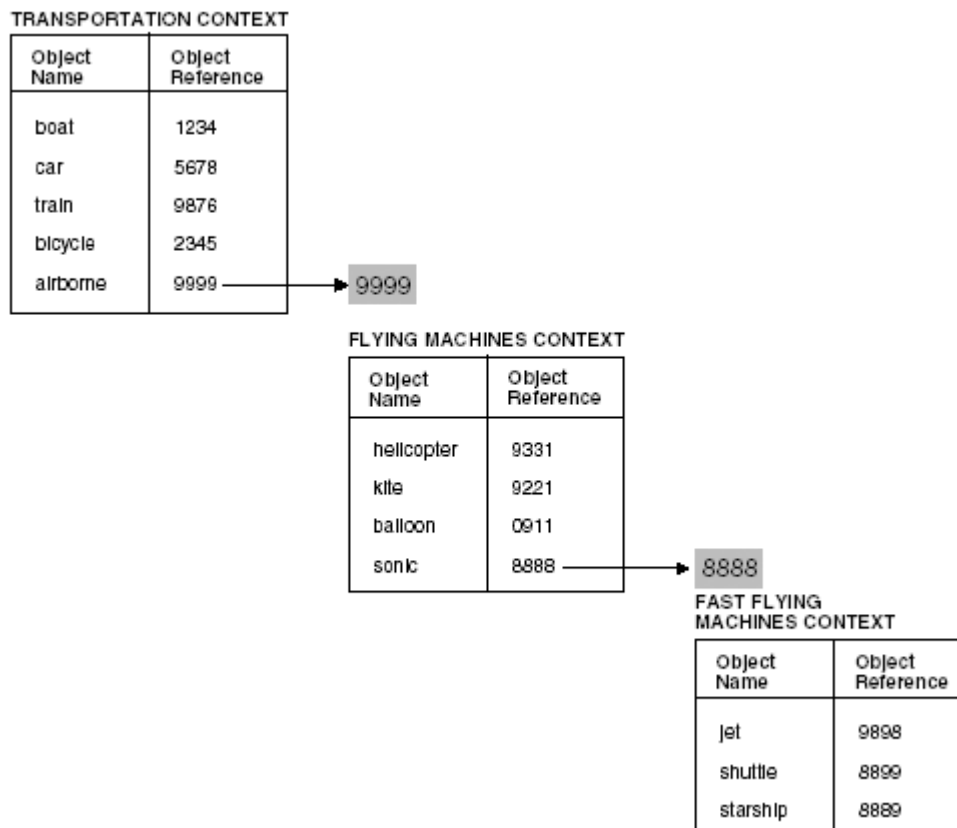
<sup>[4]</sup> wCorba is the CORBA standard for wireless remote object interaction. White papers and case studies for the wireless CORBA standard are available at [www.omg.org](http://www.omg.org).

## 8.6 The Naming Service

The naming service standard provides a mechanism for mapping names to object references. The requester of an IOR provides a name to the naming service and the naming service returns the object reference associated with that name.

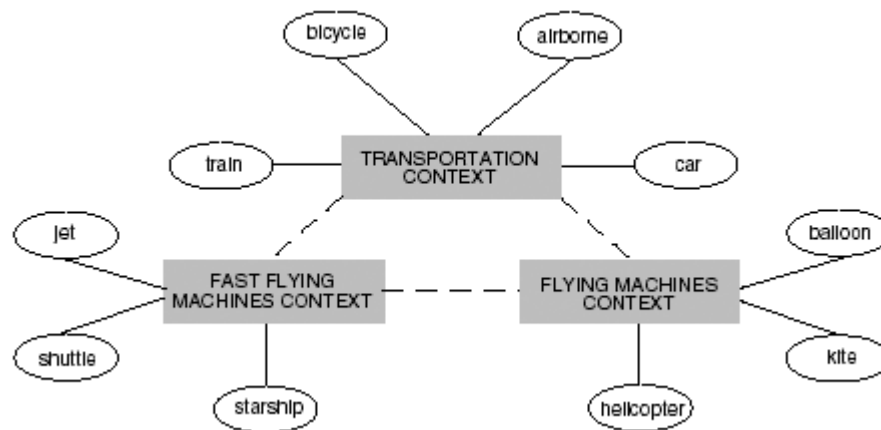
The naming service acts as a kind of telephone directory, in which the name is used to look up the number. It allows client and consumer programs to look up object references by name. The naming service can be used to map other application resources in addition to providing simple IOR maps. A mapping from a name to an object reference is called name binding. A collection of name bindings is associated with a naming context object. To illustrate the notion of a naming context, let's say we have an application that does travel planning that consists of a large and diverse collection of objects. We can organize these groups of objects according to function. Some objects are associated with file I/O, some object with security. Other objects are specifically related to transportation: train, bus, car, and bicycle objects. Each grouping forms a context. For instance, to logically group the transportation related objects together we can create a transportation context, and associate each of our forms of transportation with that context. This grouping forms a naming context. We bind the name of each form of transportation with its IOR. This is name binding. We then associate that binding with the transportation context. We use contexts to logically organize groups of related objects. Furthermore, a collection of connected naming contexts forms a naming graph. Naming contexts are represented by objects. Since a naming context is implemented as an object, it can participate in name binding just like any other object. This means that a naming context can potentially contain other naming contexts. For instance, [Figure 8-7](#) contains several contexts including a logical representation for our transportation context.

Figure 8-7. Several different naming contexts.



Notice that the last entry in the transportation context is the name airborne. The airborne name maps to another context named flying\_machines. The flying\_machines context contains bindings of several objects related by function. The transportation context, together with the flying\_machines context, form a naming graph. Notice in [Figure 8-7](#) that the last object in the flying\_machines context is named sonic. The sonic name maps to the fast\_flying\_machines context. That is, the sonic name has an object reference of 8888. This adds another context to the naming graph. This is an example of one naming context containing another naming context. The naming graph can be used to represent the "big picture" of the structure of the relationships within a distributed object-oriented application. The naming graph captures the landscape of a distributed application. For multiagent systems the naming graph can be used as a kind of semantic network (see [sidebar 8.1](#)). Although the objects involved may be scattered among diverse hardware platforms, operating systems, programming languages, and geographical locations, the naming graph can present a single logical structure of the relationships and connections between the objects. [Figure 8-8](#) shows an alternative representation of the naming graph from [Figure 8-7](#). [Figure 8-8](#) has the same naming contexts as [Figure 8-7](#) and it clearly shows the relationships between the naming contexts. [Figure 8-8](#) also demonstrates that there is a path from the transportation context to the fast\_flying\_machines context and then back to the transportation context.

**Figure 8-8. An alternative representation of the naming graph.**



Graph traversal algorithms can even be employed to traverse through the naming graph in the process of distributed problem solving. Using traversal in this way, various paths through a naming graph can represent solutions to problems. The naming service provides the requester access to naming contexts and naming graphs. Naming contexts can be accessed through naming graphs. Bindings can be accessed through naming contexts. The binding provides a direct association between a name and an object reference. [Program 8.3](#) shows a simple producer that creates a name binding and associates that name binding with a naming context.

**Program 8.3**

```

1 #include <iostream>
2 #include <fstream>
3 #include "permutation_impl.h"
4 #define MICO_CONF_IMR
5 #include <CORBA-SMALL.h>
6 #include <iostream.h>
7 #include <fstream.h>
8 #include <unistd.h>
9 #include <mico/CosNaming.h>
10
11

```

```

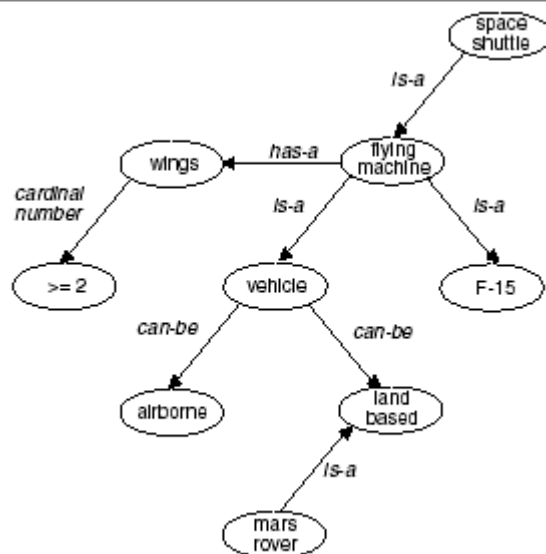
12 int main(int argc, char *argv[])
13 {
14     CORBA::ORB_var Orb = CORBA::ORB_init
        (argc,argv, "mico-local-orb");
15     CORBA::Object_var PoaObj =
        Orb->resolve_initial_references("RootPOA");
16     PortableServer::POA_var Poa =
        PortableServer::POA::_narrow(PoaObj);
17     PortableServer::POAManager_var Mgr = Poa->the_POAManager();
18     inversion Server;
19     PortableServer::ObjectId_var Oid =
        Poa->activate_object(&Server);
20     Mgr->activate();
21     permutation_ptr ObjectReference = Server._this();
22     CORBA::Object_var NameService =
        Orb->resolve_initial_references ("NameService");
23     CosNaming::NamingContext_var NamingContext =
        CosNaming::NamingContext::_narrow (NameService);
24     CosNaming::Name name;
25     name.length (1);
26     name[0].id = CORBA::string_dup ("Inflection");
27     name[0].kind = CORBA::string_dup (" ");
28     NamingContext->bind (name, ObjectReference);
29     Orb->run();
30     Poa->destroy(TRUE,TRUE);
31     return(0);
32 }
33
34

```

## S 8.1. Semantic Networks

A semantic network or semantic net is one of the oldest and easiest to understand knowledge representation schemes. A semantic network is basically a graphic depiction of knowledge that shows the hierarchical relationships between objects. [Sidebar Figure 8-1](#) shows a simple semantic network that conveys knowledge about vehicles in general and knowledge about certain vehicles in particular.

. Sidebar Figure 8-1 A simple vehicle semantic network.



The circles in the semantic net are called nodes. The lines are called links. The links represent some kind of relationship between the nodes. The nodes are used to represent objects and facts or descriptors. Links are used to represent relationships and connections. Some links are definitional while other links can be computational. The links can be used to show inheritance or subordination. Together the nodes and the links convey chunks of knowledge. For example, from the semantic network in Sidebar [Figure 8-1](#), we know that a F-15 is a vehicle and a flying machine that has at least two wings. Semantic networks are used to understand and design the knowledge needed by problem-solving software.

### 8.6.1 Using the Naming Service and Creating Naming Contexts

On line 22, the server program gets a reference to the naming service:

```
CORBA::Object_var NameService = Orb->resolve_initial_
references ("NameService");
```

In addition to returning object references for the Implementation Repository and the Interface Repository, the `resolve_initial_references()` method of the ORB is used to return a reference to the naming service. After obtaining a reference to the naming service, the server program creates a naming context from the object reference of the naming service on line 23:

```
CosNaming::NamingContext_var NamingContext =
CosNaming::NamingContext::_narrow(NameService);
```

This technique provides a naming context referred to as the initial naming context. The initial naming context plays the part of a default context. Once the naming service is located and the initial naming context is created, then the server program can add name/object reference pairs (name bindings) to the context. The names may be domain objects or other contexts. To add a name/object pair to a context, a name must first be created. Names are implemented in the CORBA standard by the NameComponent structure:

```
struct NameComponent {
    //...
    Istring_var id;
    Istring_var kind;
}
```

The MICO implementation of CORBA declaring the NameComponent structure is the CosNaming.h file. The NameComponent structure has two attributes: id and kind. The first attribute is used to hold the text of the name and the second attribute is an identifier that can be used to classify the object. For example:

```
//...
CosNaming::Name ObjectName;
ObjectName.length(1);
ObjectName.id = Corba::string_dup("train");
ObjectName.kind = Corba::string_dup("land_transportation");
NamingContext->bind(ObjectName, ObjectReference);
//...
```

Declares a NameComponent object. The id attribute is set to "train" and the kind attribute is set to land\_transportation. Obviously the id attribute should be descriptive of the object. The kind attribute can be used to describe the context or the logical group that the object belongs to. In this case, it classifies train as a land\_transportation object. The bind() method maps the ObjectName to the ObjectReference and associates it with the initial naming context. A name can consist of multiple NameComponent objects. If the name only consists of a single NameComponent, it is called a simple name. If it consists of multiple NameComponent objects, it is called a component name. If the name is a compound name, then the kind attribute can be used to describe a relationship. This technique is discussed further in [Chapter 12](#). [Program 8.3](#) binds its object with an object reference and associates it with a naming context. Once it is associated with the naming context, then the client object may access it through the name service. In [Programs 8.1](#) and [8.2](#), we used a file to communicate a stringified IOR between the consumer program and the producer program. The naming service is used for communication with the client for [Program 8.3](#).

The details for installing and executing the naming service is implementation specific. The MICO environment contains a program named nsd that implements a COS-compliant naming service. The nsd program requires the micod daemon to be running and appropriate entries to be made to the Implementation Repository before the naming service will be available to the consumer program. See the man pages for nsd, micod, and imr for a description of these programs and the MICO manual for a description of how they are used. Furthermore, the MICO distribution is accompanied by a wealth of examples of how to use the imr, nsd, micod, and ird programs. [Example 8.5](#) is an excerpt from the shell script used to set up the server in [Program 8.3](#) so that the name service would be available to the consumer program.

**Example 8.5 Shell script that adds an entry to the Implementation Repository and starts the naming service.**

```
micod -ORBIIOPAddr inet:hostname:portnumber -forward &
imr create NameService poa 'which nsd' IDL:omg.org/CosNaming/
NamingContext:1.0#NameService \
    -ORBImpRepoAddr inet:hostname:portnumber \
    -ORBNamingAddr inet:hostname:hostname:portnumberportnumber

imr create permutation persistent "'pwd'/permutation_server \
    -ORBImpRepoAddr inet:hostname:portnumber \

    -ORBNamingAddr inet:hostname:portnumber" IDL:permutation:1.0 \
    -ORBImpRepoAddr inet:hostname:portnumber \
    -ORBNamingAddr inet:hostname:portnumber
imr activate permutation -ORBImpRepoAddr inet:hostname:portnumber \
    -ORBNamingAddr inet:hostname:portnumber
```

This shell script can be used in conjunction with the server in [Program 8.3](#). In fact, this script actually

helps to automatically start the server program named `permutation_server`. Note that `hostname` and `portnumber` in [Example 8.5](#) need to be replaced by the `hostname` of the computer where the server is running and an appropriate port number.

## 8.6.2 A Name Service Consumer/Client

[Program 8.3](#) associates the name of an object with a naming context. [Program 8.4](#) contains a consumer program that uses the naming service to access the object/reference bindings that were created in [Program 8.3](#). [Program 8.3](#) produces a permutation of any string of characters that it receives. Permutations are created by inflections of the characters within a string. For instance:

Objcte	JbOetc	tbOjec
Ojbect	JObetc	
Ojbcet	JtObec	

are permutations of the string `Object`. The client gives the server a string to permute and the server generates `N` permutations. The server associates the name "Inflection" with the naming context. This name is the name that the client program will have to specify in order to get the object reference from the naming context.

### Program 8.4

```

1  int main(int argc, char *argv[])
2  {
3
4  try{
5      CORBA::ORB_var Orb = CORBA::ORB_init
        (argc,argv,"mico-local-orb");
6      object_reference Remote("NameService",Orb);
7      Remote.objectName("Inflection");
8      permutation_var Client =
        permutation::_narrow(Remote.objectReference());
9      char Value[1000];
10     strcpy(Value,"Common Object Request Broker");
11     Client->original(Value);
12     int N;
13     for(N = 0;N < 15;N++)
14     {
15         cout << "Value of nextPermutation() "
            << Client->nextPermutation() << endl;
16     }
17 }
18 catch (CosNaming::NamingContext::NotFound_catch &exc) {
19     cerr << " Object NotFound exception" << endl;
20 }
21 catch (CosNaming::NamingContext::InvalidName_catch &exc) {
22     cerr << "InvalidName exception" << endl;
23 }
24
25 return(0);
26 }
```



Three steps the consumer program must take to access the appropriate object in the naming context are:

1. Get a reference to the name service.
2. Obtain a reference to the appropriate naming context through the name service.
3. Obtain a reference to the appropriate object through the naming context.

Step 1 is accomplished by calling the `resolve_initial_references()` method:

```
//...
CORBA::Object_var NameService;
NameService = Orb->resolve_initial_references ("NameService");
//...
```

This will return an object reference to the name service. In Step 2 this reference is used to get an object reference for the naming context:

```
CosNaming::NamingContext_var NameContext;
NameContext = CosNaming::NamingContext::_narrow (NameService);
```

The value of `NameService` is narrowed in Step 3, resulting in an object reference for `NameContext`. The consumer program needs the `NameContext` object so that it may call the `NameContext`'s `resolve()` method. The technique from [Program 8.3](#) lines 24-27 is used to construct the name that will be passed to the `NameContext`'s `resolve()` method:

```
Name.length (1);
Name[0].id = CORBA::string_dup ("Inflection");
Name[0].kind = CORBA::string_dup (" ");
try {
    ObjectReference = NameContext->resolve (Name);
}
```

The `resolve()` method will return the object reference associated with the name. In this case, the object's name is "Inflection." Note that this is the same name associated with the naming context on line 28 from [Program 8.3](#). Once the consumer program has this object reference, it can be narrowed and then the remote object can be accessed by the consumer program. The process of obtaining an object reference for a remote object is such a common event that it makes sense to simplify the process by encapsulating the components within a class.

```
class object_reference{
//...
protected:
    CORBA::Object_var NameService;
    CosNaming::NamingContext_var NameContext;
    CosNaming::Name Name;
    CORBA::Object_var ObjectReference;
public:
    object_reference(char *Service,CORBA::ORB_var Orb);
    CORBA::Object_var objectReference(void);
    void objectName(char *FileName,CORBA::ORB_var Orb);
    void objectName(char *OName);
//...
}
```

[Program 8.4](#) takes advantage of the simple skeleton `object_reference` class that we have created for this purpose.

Notice on line 6 from [Program 8.4](#) that an object named Remote of type object\_reference is created. On line 8, this object is used to obtain a reference to the remote object using the method call:

```
Remote.objectReference();
```

After making this call the consumer program has access to the remote object. The object\_reference class hides some of the work that needs to be done and therefore makes writing the consumer program easier. The constructor for the object\_reference class is called on line 6 of [Program 8.4](#). The constructor is implemented as:

```
object_reference::object_reference(char *Service,CORBA::ORB_var Orb)
{
    NameService = Orb->resolve_initial_references (Service);
    NameContext = CosNaming::NamingContext::_narrow (NameService);
}
```

The constructor gets a reference to the name service and instantiates the NameContext object. On line 7, the object's name is passed to the method objectName(). This process will use the naming context to retrieve the object reference associated with the object's name. The objectName() method is implemented as:

```
void object_reference::objectName(char *OName)
{
    Name.length (1);
    Name[0].id = CORBA::string_dup (OName);
    Name[0].kind = CORBA::string_dup (" ");
    try {
        ObjectReference = NameContext->resolve (Name);
    }
    catch(...){
        cerr << "    Problem resolving Name " << endl;
        throw;
    }
}
```

After the objectName() method is called the consumer program has access to the remote object's reference. All that is left to do is to call the objectReference() method. This occurs on line 8 of [Program 8.4](#). The resolve() function does most of the work in the objectName() method. [Programs 8.3](#) and [8.4](#) form a simple distributed client/server application that uses the naming service instead of stringified IORs to communicate object references. Both the naming service approach and the stringified IOR can be used in an intranet or on the Internet. Both can be used as support structure components within the context of the new Web services model.

## 8.7 A Closer Look at Object Adapters

In addition to the name service and naming context object, the server in [Program 8.3](#) also uses a portable object adapter. Recall from [Figure 8-6](#) that the adapter acts as a kind of middleman between the ORB and the servant object that actually does the work of the CORBA object. We can compare a servant object to a ghost writer that writes a book on behalf of a celebrity. The publicists, marketers, and lawyers interact with the celebrity. The celebrity gets all the credit, but the ghost writer does the actual work and writing involved. The CORBA object publishes an interface to the outside world and is the celebrity in a CORBA program. The client or producer program interacts with the interface that the CORBA object provides, however, it's the servant object playing the part of the ghost writer that actually does the real work. The servant object has its own protocol. This protocol might be different from the one presented by the CORBA object. The CORBA object might present a C++ interface to the client. The servant object might be implemented in Java, Smalltalk, Fortran and so on. The object adapter provides an interface to the servant object. It adapts the interface so that the implementation of the servant object is transparent to the ORB and the client program. A CORBA implementation will normally have support for two types of object adapters: the Basic Object Adapter (BOA) and the Portable Object Adapter (POA). The BOA was the original adapter specified by the CORBA standard. The POA was designed to replace the BOA and is considerably more flexible and most commonly used. The BOA is a barebones adapter that has minimal capabilities. However, the BOA can be used to activate object implementations based on information stored in the Implementation Repository. [Table 8-4](#) contains some of the commonly found elements in an Implementation Repository.

The BOA uses the activation mode and the path from the Implementation Repository to start the execution of a producer or server object. Although some of the simpler examples in this chapter used the BOA, we recommend that you use the POA for any serious CORBA development. The POA:

- Supports transparent object activation
- Supports transient objects
- Supports implicit activation of servant objects
- Supports persistent objects across server boundaries

**Table 8-4. Some Commonly Found Elements in an Implementation Repository**

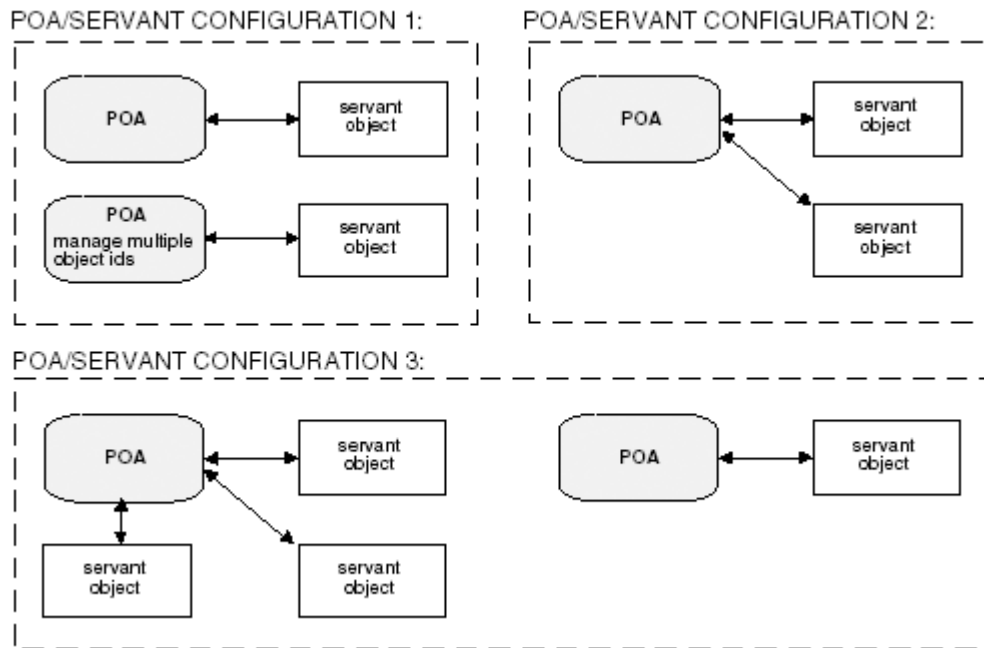
<b>Implementation Repository Elements</b>	<b>Description</b>
object name	Unique identifier for each object.
activation mode	Shared, unshared, persistent, permethod library.
path	Name and path of the binary.
list of repository IDs	

Perhaps the most important function of the POA is to interact with servant objects. The CORBA specification defines a servant accordingly:

A servant is a programming language object or entity that implements requests on one or more objects. Servants generally exist within the context of a server process. Requests made on an object's references are mediated by the ORB and transformed into invocations on a particular servant. In the course of an object's lifetime it may be associated with multiple servants.

Every servant object will have at least one POA. However, other configurations are possible. [Figure 8-9](#) shows the configuration possibilities between POAs and servants.

**Figure 8-9. Configuration possibilities between POAs and servants.**



POAs are managed in part by POA manager objects. The CORBA specification defines a POA manager accordingly.

A POA manager is an object that encapsulates the processing state of one or more POAs. Using operation on a POA manager, the developer can cause requests for the associated POAs to be queued or discarded. The developer can also use the POA manager to deactivate the POAs.

The server in [Program 8.3](#) provides a simple example of how to use POAs and POA manager objects. A complete discussion of the POA is beyond the scope of this book. For a thorough discussion of POAs, see *Advanced CORBA Programming with C++* by Michi Henning and Steve Vinoski. The MICO distribution also contains several examples of how to use some of the more advanced features of the POA.

## 8.8 Implementation and Interface Repositories

The ORB uses the Implementation Repository to locate objects when stringified IORs are not available. Implementation Repositories are normally ORB specific and are used as a convenient place to store environment-specific information (e.g., security information, debugging information, etc.). The Implementation Repository will contain enough information to allow the ORB to locate the object's path and binary executable. The imr tool is used with MICO distributions to manage the Implementation Repository. The imr tool is used to display, list, add, and delete entries from the Implementation Repository. For example:

```
imr create permutation persistent " 'pwd'/permutation_server \  
    -ORBImplRepoAddr inet:hostname:portnumber \  
    -ORBNamingAddr inet:hostname:portnumber" IDL:permutation:1.0 \  

```

Adds an entry to the repository with the name permutation. The executable file is located at 'pwd'/permutation\_server. The entry also includes the hostname and the portnumber that the program should be executed on. The Implementation Repository is a good place to store this type of information about an object. This entry is also given the mode of persistent. The ORB uses the information in this entry to properly initiate the execution of the program named permutation\_server. See the man pages for a complete list of options available for the imr tool. The Interface Repository is used in addition to the Implementation Repository to store runtime information about each object. The Interface Repository can be used to discover the interface to CORBA objects dynamically because the IDL information about CORBA objects can be stored in the Interface Repository. The ird tool implements the Interface Repository for MICO distributions of CORBA. Although the CORBA specification describes the logical features of the Implementation Repository and the Interface Repository, the specifics are environment, distribution, and vendor specific. Also, the manner in which information is placed into and managed within an Implementation and Interface Repository will be vendor specific.

## 8.9 Simple Distributed Web Services Using CORBA

The addresses for Implementation Repositories and naming services can be imbedded within HTML and used as part of a CGI call to a Web server. This technique can be used to implement simple distributed Web services using CORBA. [Example 8.6](#) shows a simple HTML entry. When the link is clicked, a CORBA client executes. The CORBA client can then get to the server using the address of the Implementation Repository and the naming service that was passed from the HTML CGI command.

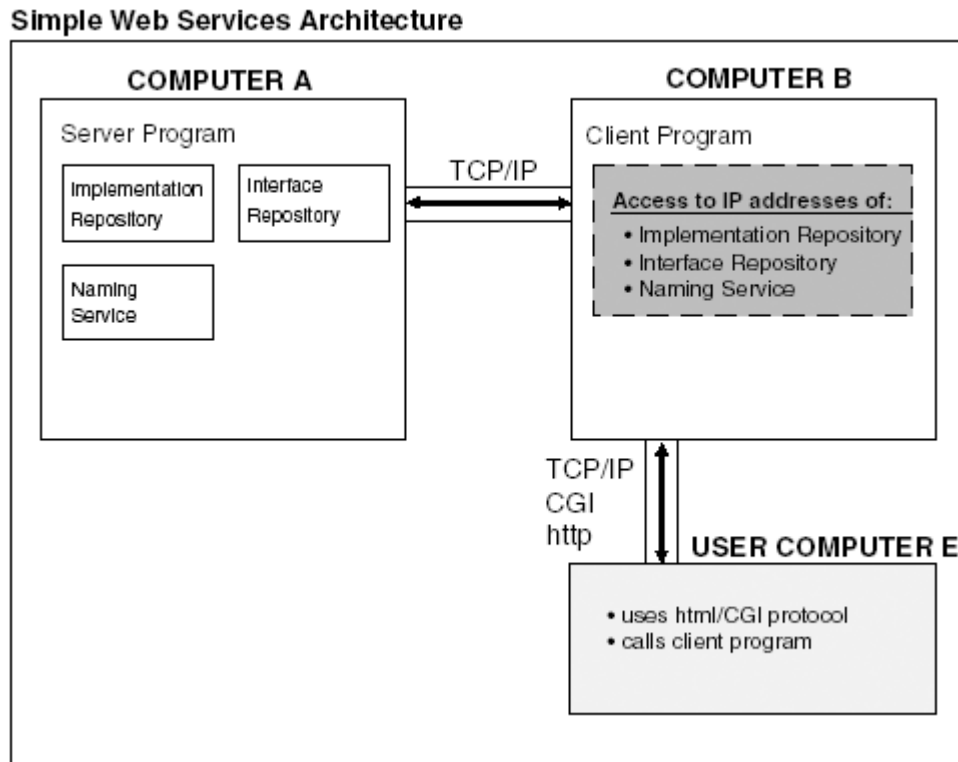
**Example 8.6 A HTML document with an embedded call to a CORBA client program.**

```
<HTML>  
<HEAD>  
<TITLE> CORBA</TITLE>  
</HEAD>  
<BODY>  
<a href="http://www.somewhere.org/cgi-bin/client?-  
ORBImplRepoAddr+inet:hostname:port+  
ORBNamingAddr+inet:hostname:port">Click</a>  
<P>  
</HTML>
```

Here the client refers to a program that will access a CORBA producer or server program. The client has the name of the object that needs to be accessed and uses the naming service to resolve the reference. This technique does not require code to be downloaded to the user's computer. Instead, the client code is executed on the Web server and will access the CORBA-based server program whether it

is on an intranet connected to a Web server or somewhere else on the Internet. The client program will respond to the HTML browser using the appropriate CGI protocol. [Figure 8-10](#) shows a simple Web services configuration using CORBA components.

**Figure 8-10.** A simple Web services configuration using CORBA components.



In addition to http, telnet can be used to launch CORBA-based clients and servers. The http protocol and the telnet protocol can be used to support global distribution of CORBA components. It is important to remember software, data, and system security when considering the design of distributed components that will be used across the Internet or intranet. Although security implementations and requirements are beyond the scope of this book, we mention it as a fundamental consideration in any distributed design. The Implementation Repository can be used to store security-type information. A CORBA implementation can be used in conjunction with SSL (Secure Socket Layer) and SSH (Secure Shell).

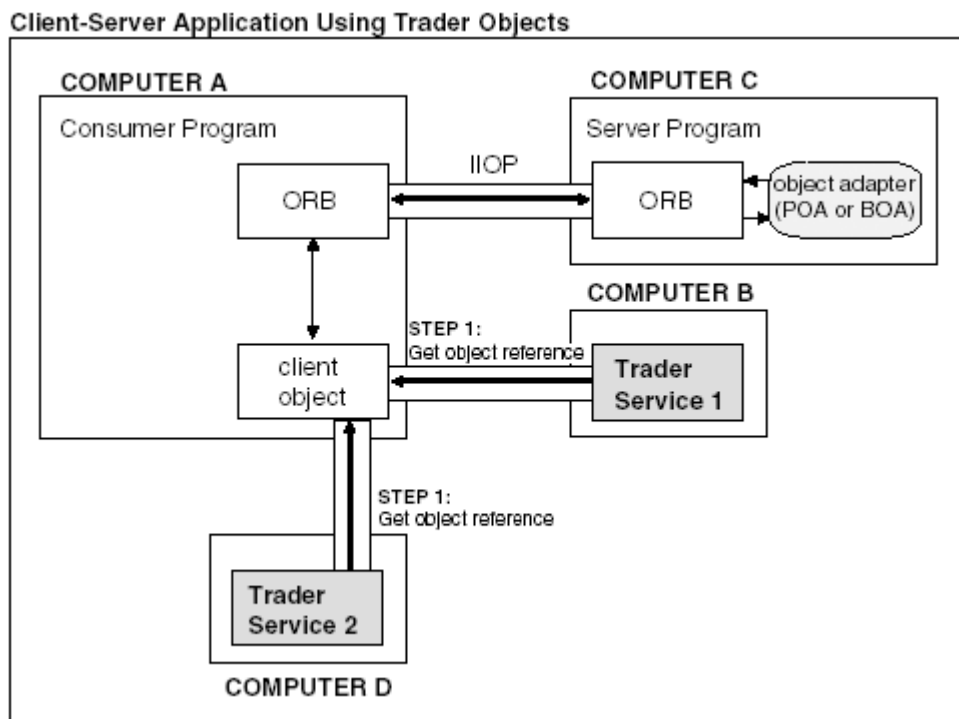
## 8.10 The Trading Service

In addition to stringified IORs and the naming service, the CORBA specification includes a more advanced and dynamic method of obtaining object references called the trading service. The trading service offers a more discovery-based approach to interacting with remote objects. Instead of interacting with a naming service, the client interacts with a trader. A trader has access to object references in the same manner as a naming service. However, the trader associates descriptions, and interfaces with the object references instead of a simple name. Whereas the naming service contains name/reference pairs, the trader contains descriptions-interfaces/reference pairs. Clients can describe the object they are looking for to the trader and the trader responds with an object reference if a match is found. This is a very powerful search method. Not only can the client be unaware of the object's location, it can also be unaware of the object's name. This allows the client to query a trader based on a list of services that it needs instead of looking for a particular object. This allows the client to have a I-don't-care-who-or-where approach. The CORBA specification defines a trader accordingly:

A trader is an object that supports the trading object service in a distributed environment. It can be viewed as an object through which other objects can advertise their capabilities and match their needs against advertised capabilities. Advertising a capability or offering a service is called "export." Matching against the needs or discovering services is called "import." Export and import facilitate dynamic discovery of, and late binding to, services.

In the same way that connecting two or more naming contexts produce naming graphs, connecting two or more traders produce trading graphs. Naming and trading graphs are powerful methods of knowledge and capability representation. Naming graphs and trading graphs provide the foundations for global Web services and telnet services. Traversing naming and trading graphs might include hops that have the potential to visit anywhere on the local network, intranet, extranet, or the Internet. Like naming contexts, traders typically represent certain kinds of objects. For instance, we might have some traders that have access to credit card objects while other traders have access to compression and encryption objects. We can have traders that deal in weather and geography objects. We can have traders that deal in financial and insurance services. If each of these traders were linked, we would have a trading graph. If one trader trades on behalf of other traders, we would have what is known as a trade federation. When a client describes the services that it needs to one trader and that trader then contacts other traders to locate the required services, the client and the trader are involved with a trade federation. This is the most powerful and flexible form of I-don't-care-who-or-where request that the client can perform. When a trade federation returns an object reference it can literally be from anywhere and may be implemented by a servant object(s) whose operating system and language are totally foreign to the client program. Federations of traders provide access to very large and diverse collections of services. Keep in mind that the CORBA standard includes a wireless specification wCORBA. This has tremendous implications for the design of mobile agent and multiagent systems. [Figure 8-11](#) shows the basic architecture of a CORBA-based client/server application that makes requests of traders.

**Figure 8-11. Basic architecture of a CORBA-based client-server application that makes requests of traders.**



A client program may interact directly with a trader or traders or indirectly with a trader through the federation. Notice in [Figure 8-11](#) that the object reference is obtained and then the interaction with the ORB occurs. [Table 8-5](#) shows common terms used with trader programming.

**Table 8-5. Common Trader Programming Terms**

<b>Trader Programming Terms</b>	<b>Description</b>
Exporter	Advertises a service with a trader. An exporter can be the service provider or it can advertise a service on behalf of another.
Importer	Uses a trader to search for services matching some criteria. An importer can be the potential service client or it can import a service on behalf of another.
Service offer	Contains a description of the service being advertised. It contains a service type name, object reference, and object properties.

## **8.11 The Client/Server Paradigm**

The terms "client" and "server" are applied in many ways to many different kinds of software applications. The client/server paradigm divides a pattern of work between two parties represented by either processes or threads. One party, the client, makes requests for data or for action. The other party, the server, fulfills the requests. The roles of requester and request satisfier are common themes found in many different software applications. The terms client/server are used at the operating-system level to describe many producer-consumer relationships that can occur between processes. For instance, when a fifo is used to connect two processes, one of the processes takes on the role of server and the other of client. Sometimes a client can take on the role of server when it receives requests. Similarly, a server may take on the role of client when it needs to make requests of another program. The client/server configuration is the most fundamental architecture for distributed programming. The type of server involved usually characterizes the entire application. [Table 8-6](#) shows the most commonly found types of software servers.

Blackboards and multiagent systems are the two primary architectures that we use in this book to support parallel and distributed programming. We place special emphasis on the logic server defined in [Table 8-6](#). The logic server is a special type of an application server, and is used to perform problem solving that requires intense symbolic and possibly parallel computation. The process of inference and deduction is often processor intensive and can benefit from parallel processors. Usually the more processors available to logic servers, the better. The Agent and Blackboard architectures that we discuss in [Chapter 12](#) and 13 rely on the notion of distributed logic servers that can cooperatively solve problems over a network, an intranet, or the Internet. Although the Blackboard and Agents form more of a peer-to-peer architecture, they are clients to the logic servers that they access. Distributed objects are used to implement all of the components involved. CORBA is used to facilitate the network programming.

## **Summary**

Distributed programming involves programs that execute in different processes. Each process can potentially reside on a different computer and possibly on a different network with different network protocols. Distributed programming techniques allow the developer to divide an application into separately executing modules that will either have some kind of producer-consumer relationship or peer-to-peer relationship. The modules each have their own address space and computer resources.



Distributed programming can be used to take advantage of special processors, peripherals, and other computer resources (i.e., database servers, applications servers, e-mail servers, etc.). CORBA is the standard for distributed object-oriented programming. We provide an introduction to some of the simple basics of CORBA programming. However, this chapter barely scratches the surface of the CORBA specification and CORBA services. It provides only enough to see what the basic components look like and how a simple distributed program can be constructed. The CORBA specifications for Web services, MAF, naming services, and so on, can be obtained from [www.omg.org](http://www.omg.org). Michi Henning and Steve Vionosk provide a detailed resource in their *Advanced CORBA Programming with C++*. The naming and trader graphs provide the basis for powerful distributed knowledge representation that can be used in conjunction with multiagent programming. They provide the basis for the next level of smart Web services.

**Table 8-6. Common Types of Software Servers**

Types of Software Servers	Description
Application servers	Used to provide multiple clients with access to an application. It divides work in an application between the client and the server. The majority of the work is done on the server and the client (with its own processor) performs part of the work.
File servers	Acts as a central repository for shared documents, multimedia files, databases, and so on. The clients are usually terminals or workstations on a network. The client makes requests for files or records within the files, then the file server transmits the request to the client. The file server maintains data integrity and enforces file access security.
Database servers	Splits the processing of an application between different machines in a network environment. A client makes requests for an item of data, then the database server locates the data and transmits the request to the client. The database server can process complex information queries that may require joins and intersections of multiple databases.
Transaction servers	Used to perform transactions that take place on the machine or machines that contain the transaction server. Every action or update completes in its entirety without interruption. If any problems are encountered, all actions or updates are undone and the transaction is tried again.
Logic servers	Used to perform problem solving that requires intense symbolic computation. It is able to find both implicit and explicit information within a database. The logic server is able to deduce or infer information that has not been explicitly entered into the database. It consists of a database with one or more built-in inference engines. The inference engine is used to obtain conclusions and inferences from the server. The database consists of rules, theorems, axioms, and procedures. Queries submitted to the logic server causes it to perform deduction, induction, abduction, or some combination.

## Chapter 9. SPMD and MPMD Using Templates and the MPI

"There must be an essentially non-algorithmic ingredient in the action of consciousness."

—Roger Penrose, *The Emperor's New Mind*

In this Chapter

- [Work Breakdown Structure for the MPI](#)
- [Using Template Functions to Represent MPI Tasks](#)
- [Simplifying MPI Communications](#)
- [Summary](#)

Templates support the notion of parameterized programming. The basic idea of parameterized programming is to maximize software reuse by implementing software designs in as general a form as possible. Function templates support generic procedural abstractions and class templates support generic data abstractions. Typically, computer programs are already general solutions to specific problems. A program that adds two numbers is usually designed to add any two numbers. However, if the program only performed the operation of addition, we could generalize this program by allowing it to perform different operations on any two numbers. If we want the most general program, can we stop with simply allowing it to perform different operations on any two numbers? What if the numbers are of different types, that is, complex numbers and floats? We may wish to design the program so that not only can it perform different operations on any two numbers but on different types or classes of numbers (i.e., ints, floats, doubles, complex numbers, etc.). In addition, we would like the program to perform any kind of binary operation on any pair of numbers so long as that operation is legal for those two numbers. Once we have implemented such a program, the opportunities for reuse are significant. Function and class templates give this capability to the C++ programmer. This kind of generalization can be accomplished using parameterized programming.

The parameterized programming paradigm supported by C++, combined with the object-oriented paradigm that is also supported by C++, provide some unique approaches to MPI programming. As we discussed in [Chapter 1](#), the MPI (Message Passing Interface) is a standard of communication used in implementing programs that require parallelism. The MPI is implemented as a collection of more than 300 routines. The MPI functions include everything from spawning tasks to barrier synchronization to set operations. There is also a C++ representation for the MPI functions that encapsulate the functionality of the MPI into a set of classes. However, many of the advantages found in the object-oriented paradigm are not used in the MPI library. The advantages of parameterized programming are also absent. So while the MPI has important value as a standard, it does not go a long way to simplify parallel programming. It does insulate the programmer from socket programming and many of the pitfalls of network programming. That insulation is not enough. Cluster, SMP, and MPP application programming can be made easier. The template facilities in C++, and the support for true object-oriented programming, can be used to help us accomplish this goal. In this chapter, we use templates and techniques from object-oriented programming, to simplify the basic SPMD and MPMD approaches used with MPI programming.

## 9.1 Work Breakdown Structure for the MPI

One of the advantages of using the MPI over traditional UNIX/Linux processes and sockets is the ability of an MPI environment to launch multiple executables simultaneously. An MPI implementation can launch multiple executables, establish a basic relationship between the executables, and identify each executable. In this book we use the MPICH<sup>[1]</sup> implementation of MPI. The command:

[1] All the MPI examples in this book were implemented using MPICH 1.1.2 and MPICH 1.2.4 in the Linux environment.

```
$ mpirun -np 16 /tmp/mpi_example1
```

tells the MPI runtime to launch 16 processes. Each process will execute the program named `mpi_example1`. Each process may use a different processor if the processor is available. Also, each process may be on a different machine if the MPI is run in a cluster-type environment. The processes will execute concurrently. The `mpirun` command is a shell script that is responsible for starting MPI jobs on the necessary processors. This script insulates the user from details of starting concurrent processes on different machines. Here it will launch 16 copies of the program `mpi_example1`. Although the MPI-2 standard does specify spawn routines that can be used to dynamically add programs to an executing MPI application, this technique is not encouraged. In general, the number of processes needed are created at the start of an MPI application. This means that the code is replicated N number of times during startup. This scheme easily supports the SPMD (SIMD) model for concurrency because the same program is launched simultaneously on multiple processors. The data that each program needs to work on can be determined after the programs are running. This technique of starting the same program on multiple processors also has implications if the MPMD model is desired. The work that a MPI program will do is divided between the number of processes launched on startup. Which process does what and which process works on which data is coded in the executable. The computers that can be involved in the process are listed in the `machines.arch` (`machines.Linux` in our case) file by host name. The location of this file is implementation dependent. Depending on your installation, the computers listed in the file will either be able to communicate using `ssh` or the UNIX/Linux `r` commands.

### 9.1.1 Differentiating Tasks by Rank

During the startup of the processes involved in an MPI application, the MPI environment assigns each process a rank and a communication group. The rank is stored as an int. The rank serves as a kind of process id for each MPI task. The communication group determines which processes can engage in point-to-point communications. Initially, all MPI processes are assigned to a default communication group. The members of a communication group can be changed after the application has started. After each process is started, one of the first things that it should do is determine its rank. This is done with the `MPI_Comm_rank()` routine. The `MPI_Comm_rank()` routine returns the rank of the calling process. The calling process specifies what communicator it is associated with in the first argument to the routine and the rank is returned in the second argument. [Example 9.1](#) shows how the `MPI_Comm_rank()` routine is used.

### Example 9.1 Using the MPI\_Comm\_rank() routine.

```
//...
int Tag = 33;
int WorldSize;
int TaskRank;
MPI_Status Status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&TaskRank);
MPI_Comm_size(MPI_COMM_WORLD,&WorldSize);
//...
```

The MPI\_COMM\_WORLD communicator is the default communicator that all MPI tasks are assigned upon startup. MPI tasks are grouped by communicators. The communicator is what identifies a communication group. In [Example 9.1](#) the rank is returned in the variable TaskRank. Each process will have a unique rank. Once the rank is determined, then the appropriate data may be given to that task or the appropriate code for that task to execute may be determined. For instance, in Case 1:

#### Case 1: Simple MPMD

```
if(TaskRank == 1){
    // do something
}

if (TaskRank == 2){
    // do something else
}
```

#### Case 2: Simple SIMD

```
if(TaskRank == 1){
    // use this data
}

if(TaskRank == 2){
    // use that data
}
```

the rank is used to differentiate which process will do which work and in Case 2 the rank is used to differentiate which data each process will work on. Although each MPI executable starts out with the same code, MPMD (MIMD) may be achieved by using the rank and performing a branch. Likewise, once the rank is determined, data types may be assigned to the data of a process or specific data that a given process needs to work with may be identified. The rank is also used in message passing. MPI tasks identify each other in a communication exchange by ranks and communicators. The MPI\_Send() and MPI\_Recv() routines use rank for destination and source, respectively. The call:

```
MPI_Send(Buffer,Count,MPI_LONG,TaskRank,Tag,Comm);
```

will send Count number of longs to a MPI process with rank = TaskRank. The Buffer is a pointer to the data to be sent to the process TaskRank. Count represents the number of items in the Buffer, not the size of Buffer. Each message has a tag. The tag can be used to differentiate one message from another, to group messages into classes, to associate certain messages with certain communicators, and so on. The tag is an int and its value is user-defined. The Comm parameter represents the communicator that the process is assigned to or associated with. If the rank and communicator of a task are known, then messages may be sent to that task. The call:

```
MPI_Recv(Buffer,Count,MPI_INT,TaskRank,Tag,Comm,&Status);
```

will receive Count ints from a process with rank = TaskRank. This routine will cause the caller to block until it receives a message from a process with TaskRank and the appropriate value for Tag. The MPI does support wild-cards for the rank and tag parameters. These wildcards are MPI\_ANY\_TAG and MPI\_ANY\_SOURCE. If these values are used, the calling process will accept the next message that it receives regardless of the source and tag of that message. The Status parameter is of type MPI\_Status.

Information about the receive operation can be retrieved from the Status object. Three fields contained in status are MPI\_SOURCE, MPI\_TAG, and MPI\_ERROR. Therefore, the Status object can be used to determine what the tag and source of the sending process were. Once the processes know how many processes are involved, they can determine who to send messages to and who to receive messages from. Naturally, which task receives messages and which task sends messages will depend on the application. How the work is divided up between the processes will also be application dependent. Another piece of information that is determined immediately by each process before the work starts is how many other processes are involved in the application. This is done by a call to:

```
MPI_Comm_size(MPI_COMM_WORLD,&WorldSize);
```

This routine determines the size of the group of processes associated with a particular communicator. In this case, the communicator is the default communication MPI\_COMM\_WORLD. The number of processes involved are returned in the WorldSize parameter. This parameter is an int. Once each process has the WorldSize, it knows how many processes are associated with its communicator and what its rank is relative to the other processes.

### 9.1.2 Grouping Tasks by Communicators

In addition to ranks, processes are also associated with communicators. The communicator specifies a communication domain for a set of processes. All processes with the same communicator are in the same communication group. The work that a MPI program does can be divided between communicator groups. MPI\_COMM\_WORLD is the default communicator group that all processes are in initially. MPI\_Comm\_create() can be used to create new communicators. [Table 9-1](#) shows a list of short descriptions for the routines used to work with communicators.

Through the use of the rank and the communicator, MPI tasks are identified and differentiated. The rank and the communicator allow us to structure a program as SPMD or MPMD or some combination. We use the rank and the communicator in conjunction with parameterized programming and object-oriented techniques to simplify the code written for a MPI program. The templates can accommodate not only the different data aspect of SIMD but different data types may also be specified using templates. This greatly simplifies the many computational-intensive applications that do the same work but with different data types. We recommend runtime polymorphism (supported by objects), parametric polymorphism (supported by templates), function objects, and predicates to achieve MPMD (MIMD). These techniques are used in conjunction with the rank and the communicator of a MPI process to accomplish the division and assignment of work in an MPI application. When using an object-oriented approach, the work of a program is divided between families of objects. The families of objects are each associated with different communicators. Associating families of objects with different communicators helps with modularity in the design of an MPI application. This kind of division also helps with understanding how the parallelism can be applied. We have found that the object-oriented approach makes MPI programs more extensible, maintainable, and easier to debug and test.

### 9.1.3 The Anatomy of an MPI Task

[Figure 9-1](#) contains a skeleton MPI program. The tasks involved in this MPI program simply report their ranks to the MPI task whose rank == 0.

Figure 9-1. A MPI program.

```

#include <mpi.h>  ← MPI header file
int Dest;
int Tag = 50;
int WorldSize;
int TaskRank;
string M;
char MessageIn(1000);
int N;
stringstream Buffer;
MPI_Status Status;
MPI_Init(&argc,&argv); ← initialization routine
MPI_Comm_rank(MPI_COMM_WORLD,TaskRank); ← Get task rank
MPI_Comm_size(MPI_COMM_WORLD,&WorldSize); ← Get # of MPI tasks
Dest = 0; ← Tells who receives the message
N = 1;
if(TaskRank != 0){
    Buffer << "Ready To Work From Rank#" << TaskRank << endl;
    getline(Buffer,M);
    MPI_Send(const_char<char">(M.data()),M.size()+1,MPI_CHAR, Dest,Tag,MPI_COMM_WORLD); ← Send message
}
else{
    do{
        cout << "From Supervisor" << endl;
        MPI_Recv(MessageIn,100,MPI_CHAR,N,Tag,MPI_COMM_WORLD,&Status); ← Receive message
        cout << MessageIn << endl;
        cout << "Received From " << MessageIn << endl;
        N++;
    } while(N < WorldSize);
}
MPI_Finalize(); ← Finish up MPI
}

```

Every MPI program should at least have MPI\_Init() and MPI\_Finalize(). The MPI\_Init routine initializes the MPI environment for the calling task. The MPI\_Finalize() routine deallocates resources from the MPI task.

Table 9-1. Routines Used to Work with Communicators

**MPI Communicator Routines #include "mpi.h"**

<pre> int MPI_Intercomm_create (MPI_Comm LocalComm,  int LocalLeader,  MPI_Comm PeerComm,  int remote_leader,  int MessageTag,  MPI_Comm *CommOut); </pre>	<p>Creates an intercommunicator from two intracommunicators.</p>
<pre> int MPI_Intercomm_merge (MPI_Comm Comm,int High, </pre>	<p>Creates an intracommunicator from an</p>

## MPI Communicator Routines #include "mpi.h" Description

<code>MPI_Comm *CommOut);</code>	intercommunicator.
<code>int MPI_Cartdim_get (MPI_Comm Comm,int *NDims);</code>	Returns Cartesian topology information associated with a communicator.
<code>int MPI_Cart_create (MPI_Comm CommOld,int NDims, int *Dims,int *Periods, int Reorder, MPI_Comm *CommCart);</code>	Creates a new communicator to which topology information has been attached.
<code>int MPI_Cart_sub (MPI_Comm Comm, int *RemainDims, MPI_Comm *CommNew);</code>	Divides a communicator up into subgroups, which form lower dimensional Cartesian subgrids.
<code>int MPI_Cart_shift (MPI_Comm Comm, int Direction, int Display,int *Source, int *Destination);</code>	Retrieves the shifted source and destination ranks, given a shift direction and amount.
<code>int MPI_Cart_map (MPI_Comm CommOld, int NDims,int *Dims, int *Periods,int *Newrank);</code>	Maps process to Cartesian topology information.
<code>int MPI_Cart_get (MPI_Comm Comm,int MaxDims, int *Dims,int *Periods, int *Coords);</code>	Returns Cartesian topology information associated with a communicator.
<code>int MPI_Cart_coords (MPI_Comm Comm, int Rank, int MaxDims, int *Coords);</code>	Calculates process coords in Cartesian topology given rank in group.
<code>int MPI_Comm_create (MPI_Comm Comm, MPI_Group Group, MPI_Comm *CommOut);</code>	Creates a new communicator.
<code>int MPI_Comm_rank (MPI_Comm Comm,int *Rank);</code>	Calculates and returns the rank of the calling process in the communicator.
<code>int MPI_Cart_rank (MPI_Comm Comm,int *Coords,</code>	Calculates and returns the process rank in a

## MPI Communicator Routines #include "mpi.h" Description

<code>int *Rank);</code>	communicator given Cartesian location.
<code>int MPI_Comm_compare (MPI_Comm Comm1, MPI_Comm Comm2, int *Result);</code>	Compares two communicators, Comm1 and Comm2
<code>int MPI_Comm_dup (MPI_Comm CommIn, MPI_Comm *CommOut);</code>	Duplicates an already existing communicator along with all its cached information.
<code>int MPI_Comm_free (MPI_Comm *Comm);</code>	Marks the communicator object to be deallocated.
<code>int MPI_Comm_group (MPI_Comm Comm, MPI_Group *Group);</code>	Accesses the group associated with the given communicator.
<code>int MPI_Comm_size (MPI_Comm Comm,int *Size);</code>	Calculates and returns the size of the group associated with a communicator.
<code>int MPI_Comm_split (MPI_Comm Comm,int Color, int Key,MPI_Comm *CommOut);</code>	Creates new communicators based on colors and keys.
<code>int MPI_Comm_test_inter (MPI_Comm Comm,int *Flag);</code>	Determines if a communicator is an intercommunicator.
<code>int MPI_Comm_remote_group (MPI_Comm Comm, MPI_Group *Group);</code>	Accesses the remote group associated with the given intercommunicator.
<code>int MPI_Comm_remote_size (MPI_Comm Comm,int *Size);</code>	Calculates and returns the size of the remote group associated with an intercommunicator.

Every MPI task should call the `MPI_Finalize()` routine prior to exiting. Notice the calls to `MPI_COMM_rank()` and `MPI_COMM_Size()` in [Figure 9-1](#). They are used to get the rank and the number of processes that belong to an MPI application. Most MPI applications should call this function. The remaining MPI functions will depend on the application. The MPI environment supports over 300 functions. Consult your man pages for a complete listing and discussion of the MPI functions.



## 9.2 Using Template Functions to Represent MPI Tasks

Function templates allow us to generalize a procedure for any type. Let's look at a multiplication procedure that works for any data type for which multiplication is defined:

```
template<class T> T multiplies(T X, T Y)
{
    return(X * Y);
}
```

To use a template function such as this one we provide the necessary parameters for the type T. T is a stand-in for some data type that will be supplied when the template is instantiated. So we can instantiate `multiplies()` accordingly:

```
//...
multiplies<double>(3.2,4.5);
multiplies<int>(7,2);
multiplies<rational>("7/2","3/4");
//...
```

with T instantiated to double, int, and rational, thereby determining the exact implementation of the multiplication operation. Multiplication is defined differently for each data type. So the specification of the data type causes slightly different code to be executed. The template function allows us to write the `multiplies()` operation once and apply it to many different data types.

### 9.2.1 Instantiating Templates and SPMD (Datatypes)

Parameterized functions can be used with the MPI to handle situations where each process is executing the same code but is working with a different type of data. So once we have determined the TaskRank of the process, we can differentiate what data and type of data the process should work with. [Example 9.2](#) shows how to instantiate different tasks for different ranks.

**Example 9.2 Using template functions to designate what the MPI task will do.**

```
int main(int argc, char *argv[])
{
    //...
    int Tag = 2;
    int WorldSize;
    int TaskRank;
    MPI_Status Status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&TaskRank);
    MPI_Comm_size(MPI_COMM_WORLD,&WorldSize);
    //...
    switch(TaskRank)
    {
        case 1: multiplies<double>(3.2,4.6);
                break;
        case 2: multiplies<complex>(X,Y)
                break;

        //case n:

    //...
    }
```

```
}
```

Since no two tasks have the same rank, each branch in the case statement in [Example 9.2](#) will be executed by a different MPI task. Also, you may extend this type of parameterization to container arguments for template functions. This allows you to pass different containers of objects containing different types of objects to the same generic template function. For instance [Example 9.3](#) contains a generic search() template.

**Example 9.3 Using container templates as arguments to template functions.**

```
template<T> bool search(T Key,graph<T>)
{
    //...
    locate(Key)
    //...
}

//...
MPI_Comm_rank(MPI_COMM_WORLD,&TaskRank);
//...
switch(TaskRank)
{
    case 1:
    {
        graph<string> bullion;
        search<string> search("gold", bullion)
    }
    break;
    case 2:
    {
        graph<complex> Coordinates;
        search<complex>((X,Y),Coordinates);
    }
    break;
}

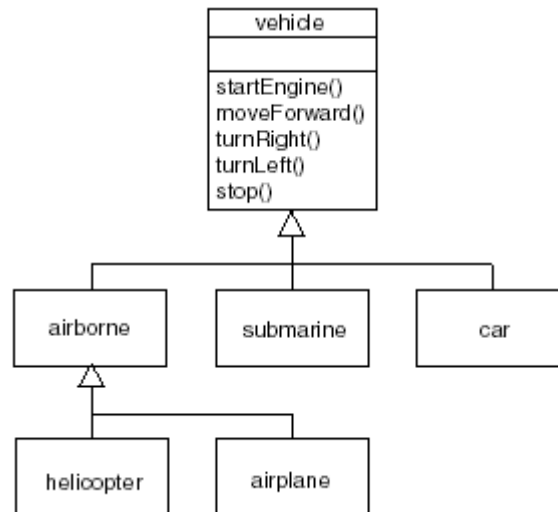
//...
```

In [Example 9.3](#), the process with TaskRank == 1 searches a graph named bullion that contains string objects and the process with TaskRank == 2 searches a graph named Coordinates containing complex numbers. We did not have to change the search() routine to accommodate the different data or data types and the MPI program is made simpler because we can reuse the search function template to search a graph container containing any type. Using templates simplifies SPMD programming. The more generic we make the MPI task, the more flexible it is. Also, once the template is debugged and tested, the reliability of all of the MPI tasks are increased since they all execute the same code.

### 9.2.2 Using Polymorphism to Implement MPMD

Polymorphism is a primary characteristic of object-oriented programming. In order for a language to support true object-oriented programming, the language must support encapsulation, inheritance, and polymorphism. Polymorphism is the ability of an object to take on many forms. Polymorphism supports the notion of "one interface, multiple implementations." A user uses one name or interface implemented in different ways by different objects. To illustrate the concept of polymorphism, let's look at a vehicle class, its descendants, and a simple function called travel() that uses the vehicle class. [Figure 9-2](#) shows the simple hierarchy for our vehicle class family.

Figure 9-2. The vehicle class family hierarchy.



Airplanes, helicopters, cars, and submarines are all descendants of type `vehicle`. A `vehicle` object can start its engine, move forward, turn right, turn left, stop, and so on. [Example 9.4](#) demonstrates how the `travel` function uses a `vehicle` object to make a computerized trip.

**Example 9.4 The `travel()` function using a `vehicle` object.**

```
void travel(vehicle *Transport)
{
    Transport->startEngine();
    Transport->moveForward();
    Transport->turnLeft();
    //...
    Transport-> stop();
}

int main(int argc, char *argv[])
{
    //...
    car *Car;
    Transportation = new Vechicle();
    travel(Car);
    //...
}
```

The `travel()` function accepts a pointer to a `vehicle` object. The `travel()` function invokes the appropriate methods of the `vehicle` object. Notice that the `main()` function in [Example 9.4](#) declares an object of type `car` and not type `vehicle`. A `car` object is passed to the function `travel()` instead of a `vehicle` object. This is possible because in C++ a pointer to a class can point to an object of that type or any objects that are descendants of that type. Since `car` inherits `vehicle`, a `vehicle` pointer can point to an object of type `car`. The function `travel()` is written without the knowledge of what types of `vehicle` object it will manipulate. The `travel()` function simply requires that its `vehicle` objects have the capability of starting an engine, moving forward, turning left and right, and so on. As long as its `vehicle` object can perform those actions, then the `travel()` function can do its work. Notice in [Figure 9-2](#) that the methods of the `vehicle` class have been declared as `virtual`. Declaring the methods as `virtual` in a base class is necessary for runtime polymorphism to work. The `car`, `helicopter`, `submarine`, and `airplane` class will each define:

```
startEngine();
moveForward();
turnLeft();
turnRight();
stop();
//...
```

relative to their type of machine. Although each type of vehicle moves forward, the method in which a car moves forward is different from the way a submarine moves forward. The way an airplane turns right is different from the way a car turns right. Therefore, each vehicle type has to implement the necessary operations to complete its class. Since these operations are declared as virtual in the base class, they are candidates for polymorphism. When the `travel()` function's vehicle pointer actually points to a car object, then the `startEngine()`, `moveForward()`, and so on called will be those methods defined in the car class. If the `travel()` function's vehicle pointer was assigned a pointer to an airplane class, then the `startEngine()`, `moveForward()`, and so on methods that belonged to the airplane class would be called. This is where the many forms, or single interface multiple implementations, come in. Although the `travel()` function only calls a single set of methods, the behavior of those methods can be radically different depending on what type of vehicle has been assigned to the vehicle pointer. In this way `travel()` is polymorphic because it may do something very different each time it is called. In fact, as long as the `travel()` function uses a pointer to a vehicle type, it may be used in the future for vehicle types that were unknown or that did not exist at the time the `travel()` function was designed. As long as the future vehicle classes inherit `vehicle` and define the necessary methods then they can be manipulated by the `travel()` function. This type of polymorphism is called runtime polymorphism. It's called runtime polymorphism because the `travel()` function does not know exactly which `startEngine()`, `moveForward()`, or `turnLeft()` functions it will call until the program is executing.

This type of polymorphism is useful when implementing MPI programs that use a MPMD model. If the work that the MPI tasks perform manipulate pointers to base classes, then polymorphism allows the MPI class to also manipulate any derived classes of the base class. If instead of pointers, the `travel()` function in [Example 9.4](#) had a declaration:

```
void travel(vehicle Transport);
```

then the `startEngine()`, `moveForward()`, and so on calls would belong to the vehicle class and there wouldn't be an easy way to manipulate derived classes. The pointer to the vehicle class and the fact that the methods in the vehicle class are declared virtual are what makes the polymorphism work. MPI tasks that manipulate pointers to base classes can take advantage of polymorphism in the same way that the `travel()` function is able to work with any kind of vehicle object present or future. This technique holds a lot of promise for the future of cluster, smp, and mpp applications that will need to implement MPMD models. To see how this MPMD works in a MPI context, let's use our `travel()` function as a MPI task that is part of a search and rescue simulation. Each MPI task is responsible for performing a search and rescue mission with a different type of vehicle object. Each vehicle will obviously have different means of mobility. Although the problem to be solved requires that each MPI task perform a search, the code is different because each task uses a different kind of vehicle object that works different and requires different data. [Example 9.5](#) would be launched in our MPICH environment using:

```
$ mpirun -np 16 /tmp/search_n_rescue
```

### Example 9.5 MPI tasks implementing simple search and rescue simulation.

```
template<T> bool travel(vehicle *Transport, set<T> Location,
                      T Object)
{
    //...
    Transport->startEngine();
    Transport->moveForward(XDegrees);
    Transport->turnLeft(YDegrees);
    //...
    if (Location.find(Transport->location() == Object){
        //... rescue()
    }
    //...
}

int main(int argc, char *argv[])
{
    //...
    int Tag = 2;
    int WorldSize;
    int TaskRank;
    MPI_Status Status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &TaskRank);
    MPI_Comm_size(MPI_COMM_WORLD, &WorldSize);
    //...
    switch(TaskRank)
    {
        case 1:
            {
                //...
                car * Car;
                set<streets> SearchSpace
                travel<streets>(Car, SearchSpace, Street);
                //...
            }
            break;
        case 2:
            {
                //...
                helicopter *BlueThunder;
                set<air_space> NationalAirSpace;
                travel<air_space>(BlueThunder, NationalAirSpace,
                                AirSpace);
                //...
            }
        //case n:
        //...
    }
}
```

This will cause search\_n\_rescue to be launched in 16 processes, with each process potentially running on a different processor and each processor potentially on a different computer. Although each process is executing the same executable, the work (code) and data that each process works with is radically

different. Templates and polymorphism are used to differentiate what each MPI task will do and what data it will use. Notice in [Example 9.5](#) that the MPI process that has a TaskRank == 1 will use a Car object to perform a search and rescue with a container that contains street objects. The MPI process that has a TaskRank == 2 will perform its simulation using helicopters and air\_space objects. Both tasks call the travel() template function. Since the travel() template function manipulates pointers to the vehicle class, it can take advantage of polymorphism and perform its operations with any descendant of type vehicle. This means that although each MPI task is calling the same travel() function, the operation that the travel() function performs will not be the same. Notice there are no case statements or if statements in the travel() function that attempt to identify what type of vehicle it is working with. The particular vehicle object it is working with is determined by the type that vehicle is pointing to. This MPI application would work with potentially 16 different vehicles, each with its own type of mobility and search space. There are other techniques that can be used to implement MPMD within a MPI environment but the polymorphic approaches generally require less code.

The two primary types of polymorphism we demonstrate are dynamic binding polymorphism supported by inheritance and virtual methods and parametric polymorphism supported by templates. The travel() function in [Example 9.5](#) uses both types of polymorphism. The inheritance-based polymorphism is demonstrated by the use of the vehicle \*Transport. The parameterized polymorphism is demonstrated by the use of set<T>, and T Object. Parametric polymorphism is the mechanism by which the same code is used on different types passed as parameters. [Table 9-2](#) lists the different types of polymorphism that may be used to simplify MPI tasks and shorten the code required to implement an MPI program.

**Table 9-2. Different Types of Polymorphism That May Be Used to Simplify MPI Tasks**

<b>Types of Polymorphism</b>	<b>Mechanisms</b>	<b>Description</b>
Runtime (dynamic)	inheritance virtual methods	All information needed to determine which function is to be executed is not known until runtime.
Parametric	templates	A mechanism in which the same code is used on different types that are passed as parameters.

### 9.2.3 Adding MPMD with Function Objects

Function objects are also used by the standard algorithms to implement a kind of horizontal polymorphism. The polymorphism implemented using vehicle \*Transport in [Example 9.5](#) is vertical because in order for it to work the classes must all be related through inheritance. When horizontal polymorphism is used, the classes are not related by inheritance but by interface. Function objects each has the operator() defined. Function objects would allow MPI tasks to be designed with the general form:

```
// function object
class some_class{
    //...
    operator();
    //
};
```

```

template<class T> T mpiTask(T X)
{
    //...
    T Result;
    Result = X()
    //...
}

```

The mpiTask() template function will then work with any type T that has the operator() function appropriately defined:

```

//...
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&TaskRank);
MPI_Comm_size(MPI_COMM_WORLD,&WorldSize);
//...

if(TaskRank == 0){
    //...
    user_defined_type M;
    mpiTask(M);
    //...
}

if(TaskRank == N){
    //...
    some_other_userdefined_type N;
    mpiTask(N);
}
//....

```

This horizontal polymorphism does not rely on inheritance or virtual functions. So if our MPI task gets its rank and then declares any type of object that has the operator() defined, then when mpiTask() is called its behavior will be dictated by whatever functionality is found in the operator() method. So although each process launched with the mpirun script is identical, the polymorphism of templates and the function objects allow each MPI task to perform different work on different data.

### 9.3 Simplifying MPI Communications

In addition to simplifying and shortening the code of the MPI task with polymorphism and templates, we can also simplify the communication between MPI tasks by taking advantage of operator overloading. The `MPI_Send()` and `MPI_Recv()` class of functions have the form:

```
MPI_Send(Buffer,Count,MPI_LONG,TaskRank,Tag,Comm);
MPI_Recv(Buffer,Count,MPI_INT,TaskRank,Tag,Comm,&Status);
```

where the calls require that the user specify the data type involved in the call and a buffer that will hold the data to be sent or received. The specification of the data type for each call of the send and receive routines can be tedious and can introduce subtle errors if the wrong types are passed. [Table 9-3](#) contains short descriptions for each of the MPI send and receive functions and their prototypes.

The goal is to make the data types and buffers as transparent as possible during send and receive operations. We would like to send and receive MPI data using the stream metaphor of the `iostream` classes. We would like to send data using syntax such as:

```
//...
int X;
float Y;
user_defined_type Z;

cout << X << Y << Z;

//...
```

Here, the developer does not have to specify the types when inserting data into `cout`. The three data types to be displayed each have the operator `<<` defined. These definitions specify how to translate the type during the insertion into the `cout` stream. Likewise, extraction from the `cin` stream:

```
//...

int X;
float Y;
user_defined_type Z;

cin >> X >> Y >> Z;
//...
```

occurs without specifying the types involved. Operator overloading allows the developer to use this technique for MPI tasks. The `cout` stream is instantiated from an `ostream` class and `cin` is instantiated from an `istream` class. These classes define the operator `<<` and `>>` for the built-in C++ data types. For instance, the `ostream` class contains a number of overloaded operator `<<` functions:

```
//...
ostream& operator<<(char c);
ostream& operator<<(unsigned char c);
ostream& operator<<(signed char c);
ostream& operator<<(const char *s);
ostream& operator<<(const unsigned char *s);
ostream& operator<<(const signed char *s);
ostream& operator<<(const void *p);
ostream& operator<<(int n);
ostream& operator<<(unsigned int n);
ostream& operator<<(long n);
ostream& operator<<(unsigned long n);
```



**Table 9-3. MPI Send and Receive Functions and Their Prototypes**

<b>MPI Send and Receive Routines #include "mpi.h"</b>	<b>Description</b>
<pre>int MPI_Send (void *Buffer,int Count,  MPI_Datatype Type,  int Destination,  int MessageTag,  MPI_Comm Comm);</pre>	Performs a basic send.
<pre>int MPI_Send_init (void *Buffer,int Count,  MPI_Datatype Type,  int Destination,  int MessageTag,  MPI_Comm Comm,  MPI_Request *Request);</pre>	Initializes a handle for a standard send.
<pre>int MPI_Ssend (void *Buffer,int Count,  MPI_Datatype Type,  int Destination,  int MessageTag,  MPI_Comm Comm);</pre>	Performs a basic synchronous send.
<pre>int MPI_Ssend_init (void *Buffer,int Count,  MPI_Datatype Type,  int Destination,  int MessageTag,  MPI_Comm Comm,  MPI_Request *Request);</pre>	Initializes a handle for a synchronous send.
<pre>int MPI_Rsend (void *Buffer, int Count,  MPI_Datatype Type,  int Destination,  int MessageTag,  MPI_Comm Comm);</pre>	Performs basic ready send.
<pre>int MPI_Rsend_init (void *Buffer,int Count,  MPI_Datatype Type,  int Destination,  int MessageTag,  MPI_Comm Comm,  MPI_Request *Request);</pre>	Initializes a handle for a ready send.
<pre>int MPI_Isend (void *Buffer,int Count,  MPI_Datatype Type,  int Destination,</pre>	Starts a nonblocking send.

MPI Send and Receive Routines #include "mpi.h"	Description
<pre>int MessageTag, MPI_Comm Comm, MPI_Request *Request);</pre>	
<pre>int MPI_Issend (void *Buffer,int Count, MPI_Datatype Type, int Destination, int MessageTag, MPI_Comm Comm, MPI_Request *Request);</pre>	Starts a nonblocking synchronous send.
<pre>int MPI_Irsend (void *Buffer, int Count, MPI_Datatype Type, int Destination, int MessageTag, MPI_Comm Comm, MPI_Request *Request);</pre>	Starts a nonblocking ready send.
<pre>int MPI_Recv (void *Buffer,int Count, MPI_Datatype Type, int source,int MessageTag, MPI_Comm Comm, MPI_Status *Status);</pre>	Performs a basic receive.
<pre>int MPI_Recv_init (void *Buffer,int Count, MPI_Datatype Type, int source,int MessageTag, MPI_Comm Comm, MPI_Request *Request);</pre>	Initializes a handle for a receive.
<pre>int MPI_Irecv (void *Buffer,int Count, MPI_Datatype Type, int source,int MessageTag, MPI_Comm Comm, MPI_Request *Request);</pre>	Begins a nonblocking receive.
<pre>int MPI_Sendrecv (void *sendBuffer, int SendCount, MPI_Datatype SendType, int Destination,int SendTag, void *recvBuffer, int RecvCount, MPI_Datatype RecvType, int Source, int RecvTag, MPI_Comm Comm,</pre>	Sends and receives a message.

**MPI Send and Receive Routines #include "mpi.h" Description**

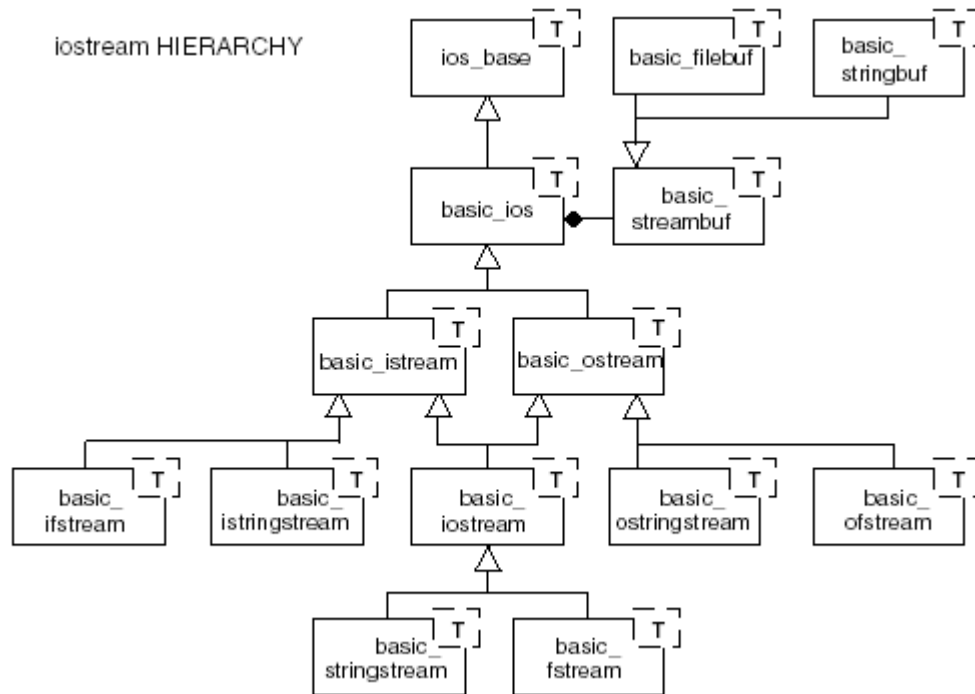
```
MPI_Status *Status);
```

```
int MPI_Sendrecv_replace
(void *Buffer,int Count,
 MPI_Datatype Type,
 int Destination,int SendTag,
 int Source,int RecvTag,
 MPI_Comm Comm,
 MPI_Status *Status);
```

Sends and receives using a single buffer.

These definitions allow the user of the ostream and the istream classes to use cout and cin objects without having to specify the data types involved in the operations. This overloading technique can be used to simplify MPI communications. We explored the idea of a PVM stream in [Chapter 6](#). Here we employ the same approach to create an MPI stream. We can use the structure of an istream and ostream as a guide for creating an mpi\_stream class. The stream classes consist of a state component, buffer component, and translation component. The state component is captured by the ios class. The buffer component is represented by the streambuf, stringbuf, or filebuf classes. The translator classes are istream, ostream, istringstream, ostringstream, ifstream, and ofstream. The state component is responsible for encapsulating the state of the stream. The format of the stream, whether the stream is in a good state or failed state, or whether the stream is at eof, and so on are captured by the ios component. The buffer components are used to hold the data that is being read or written. The translation classes translate types into streams of bytes and streams of bytes back into built-in types. [Figure 9-3](#) shows the UML class diagram for the iostream family classes.

**Figure 9-3. UML class diagram for iostream family classes.**



**9.3.1 Overloading the << and >> Operators for MPI Communication**

The relationships and functionality of the classes in [Figure 9-3](#) are used as a guideline for designing

mpi\_streams. Although going through the trouble of designing MPI stream classes is more work up front than using the MPI\_Recv() and MPI\_Send() routines directly, it will make MPI development considerably simpler in the long run. Where parallel programs can be made simpler, they should. Reducing the complexity of programs is usually a noteworthy goal. We only present a skeleton on an mpi\_stream class here. We present enough to demonstrate how the construction of an MPI stream class can be approached. Once an mpi\_stream class is designed it can be used to simplify communications in most any MPI program. [Example 9.6](#) contains an excerpt from the declaration of a mpi\_stream class.

**Example 9.6** Contains an excerpt from the declaration of a mpi\_stream class.

```
class mpios{
protected:
    int Rank;
    int Tag;
    MPI_Comm Comm;
    MPI_Status Status;
    int BufferCount;
    //...
public:
    int tag(void);
    //...

}
class mpi_stream public mpios{
protected:
    mpi_buffer Buffer;
    //...

public:
    //...
    mpi_stream(void);
    mpi_stream(int R,int T,MPI_Comm C);
    void rank(int R);
    void tag(int T);
    void comm(MPI_Comm C);
    mpi_stream &operator<<(int X);
    mpi_stream &operator<<(float X);
    mpi_stream &operator<<(string X);
    mpi_stream &operator<<(vector<long> &X);
    mpi_stream &operator<<(vector<int> &X);
    mpi_stream &operator<<(vector<float> &X);
    mpi_stream &operator<<(vector<string> &X);
    mpi_stream &operator>>(int &X);
    mpi_stream &operator>>(float &X);
    mpi_stream &operator>>(string &X);
    mpi_stream &operator>>(vector<long> &X);
    mpi_stream &operator>>(vector<int> &X);
    mpi_stream &operator>>(vector<float> &X);
    mpi_stream &operator>>(vector<string> &X);
    //...
};
```

For exposition purposes we have combined the impi\_stream and ompi\_stream class into a single mpi\_stream class. In the same manner that the istream and ostream classes overload the << and >> operators, we overload those operators as well. [Example 9.7](#) shows these overloaded operators can be defined:

**Example 9.7 Definition of << and >> operators.**

```
//...
mpi_stream &operator<<(string X)
{
    MPI_Send(const_cast<char*>(X.data()),X.size(),MPI_CHAR,Rank,Tag,Comm);
    return(*this);
}
// Over simplification of buffer
mpi_stream &operator<<(vector<long> &X)
{
    long *Buffer;
    Buffer = new long[X.size()];
    copy(X.begin(),X.end(),Buffer);
    MPI_Send(Buffer,X.size(),MPI_LONG,Rank,Tag,Comm);
    delete Buffer;
    return(*this);
}

// Over simplification of buffer
mpi_stream &operator>>(string &X)
{
    char Buffer[10000];
    MPI_Recv(Buffer,10000,MPI_CHAR,Rank,Tag,Comm,&Status);
    MPI_Get_count(&Status,MPI_CHAR,&BufferCount);
    X.append(Buffer);
    return(*this);
}
```

The `mpios` class in [Example 9.7](#) serves a similar purpose to that of the `ios` class for the `iostream`. The purpose of the `mpios` class is to maintain the state of the `mpi_stream` classes. Each data type that will be used within your MPI applications should have the operators `<<` and `>>` overloaded for them. Here, we show a few simple overloaded operators. In each case we present an over-simplification of the buffer management. In practice, exception handling and memory allocation issues are handled by template classes and allocator classes. Notice in [Example 9.7](#) that the `mpios` class holds the communicator, status of the `mpi_stream`, the buffer count, and the value for rank and tag. This is only one possible configuration for a `mpi_stream` class—there are many others. Once an `mpi_stream` class is defined it can be reused in any MPI program. Communication between MPI tasks can be written as:

```
//...
int X;
float Y;
vector<float> Z;
mpi_stream Stream(Rank,Tag,MPI_WORLD_COMM);
Stream << X << Z;
Stream << Y;
//...
Stream >> Z;
```

This notation allows the programmer to maintain the stream metaphor and simplifies the MPI code. Of course the appropriate error checking and exception handling must be included within the definitions of the `<<` and `>>` operators.

## Summary

Implementations of the SPMD and MPMD models of concurrency have much to be gained by using templates and taking advantage of polymorphism. While the MPI does include bindings for C++ it does not take advantage of object-oriented programming techniques. This presents an opportunity and challenge to developers using the MPI standard. Inheritance and polymorphism can be used to simplify MPMD programming. Parameterized or genericity programming that is supported using the template facilities of C++ can be used to simplify SPMD programming with the MPI. Dividing a program's work between objects is a natural way to discover and exploit the parallelism within an application. Families of objects can be associated with communicators to facilitate communication in the MPI between multiple groups that have different work responsibilities. Operator overloading can be used to maintain a stream metaphor with the MPI. Using object-oriented programming techniques and parameterized programming techniques within the same MPI application is a multiparadigm approach that simplifies and in most cases shortens the code. It leads to programs that are easier to debug, test, and maintain. MPI tasks implemented by template functions tend to be more reliable across different data types than separately defined functions that have to perform type casting.

## Chapter 10. Visualizing Concurrent and Distributed System Design

"Unnamed thinking which I would like to suggest may be common with us. Our brainwaves are very often wordless. We often quite suddenly perceive correct solutions to problems with which we have been striving for a long time before we have decided we will name them in one language or another. ... Very many ideas come to us in wordless form . . ."

—O. Koehler, *The Ability of Birds to Count*

In this Chapter

- [Visualizing Structures](#)
- [Visualizing Concurrent Behavior](#)
- [Visualizing the Whole System](#)
- [Summary](#)

A model of a system is the body of information gathered for the purpose of studying the system so it can be better understood by the developers and maintainers of the system. When a system is modeled, the boundaries and identification of the entities, attributes, and the activities performed by the system can be determined. Modeling is an important tool in the design process of any system. It is essential that developers fully understand the system they are developing. Modeling can reveal the concurrency embedded in the system and where distribution can be appropriately applied.

The UML (United Modeling Language) is a graphical notation used to design, visualize, model, and document the artifacts of a software system. It is the de facto standard for communicating and modeling object-oriented systems. The modeling language uses symbols and notations to represent the artifacts of a software system from different views and different focuses. The UML brings together the approaches of Grady Booch, James Rumbaugh, and Ivar Jacobson's object-oriented analysis and design methods developed in the 1980s and 1990s. It was adopted by the OMG (Object Management Group), an international organization consisting of software developers and information system vendors with over 800 members. The adoption and conformance to the UML give software developers a consistent language and tool for object analysis, specification, visualization, and documentation.

In this chapter, we show you how to visualize and model your concurrent and distributed system using the UML. Besides helping you in the design of your system, modeling will help you identify where concurrency emerges, when synchronization and communication are needed, and how and where objects can be distributed. We discuss diagramming techniques used to visualize and model concurrent systems from the structural and behavioral perspectives. Please note the classes, objects, and systems used as examples in this chapter are used for exposition purposes and may or may not necessarily reflect actual classes, objects, or structures used in an actual system.

## 10.1 Visualizing Structures

The structural view of a system focuses on the static parts of that system. This view examines how the elements in the system are constructed. It examines its attributes, properties, and operations along with its organization, composition, and relationship with other elements in the system. The diagramming techniques discussed in this section are the ones used to model:

- class, objects, templates, processes, and threads
- organization of objects that work together

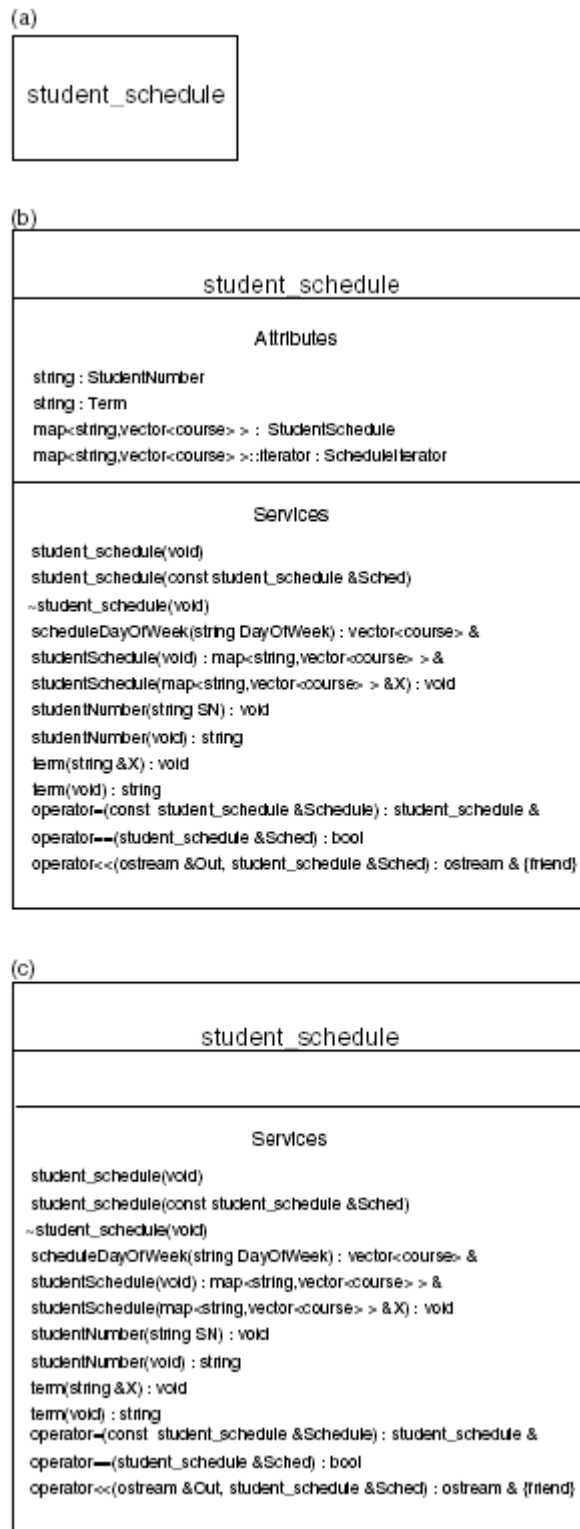
The elements documented can be conceptual or physical.

### 10.1.1 Classes and Objects

A class is a model of a construct with its attributes and behaviors. It is a description of a set of things or objects that share the same attributes. A class is the basic component of any object-oriented system. Classes can be used to represent real-world, conceptual, hardware, and software constructs. A class diagram is used to represent the classes, the objects, and the relationships that exist between them within your concurrent and/or distributed system. The class diagram is used to show the attributes and the services a class provides and the restrictions that apply to the manner in which these classes/objects are connected.

The UML provides a graphical representation of a class. The simplest representation of a class is a rectangular box that contains the name of the class. The name alone is the simple name. The class diagram can also show the attributes and services provided to the user of the class. To include attributes and services, a rectangle is drawn displaying three horizontal compartments. The top compartment displays the simple name of the class, the middle compartment displays the attributes, and the bottom compartment displays the services. The attributes and services compartments can be labeled "attributes" and "services," respectively, in order to identify each compartment. Besides the name of the class, if the attributes or services are to be shown, then the other compartment is displayed as empty. [Figure 10-1](#) shows the various ways a class can be represented.

Figure 10-1. The various ways to represent a class.



In [Figure 10-1](#), the class `student_schedule` is represented. [Figure 10-1\(a\)](#) shows the class in its simplest representation, (b) shows the class name and its attributes and services, and (c) shows the class name and its services. The compartment that contains the attributes is empty in order to communicate that the class has attributes but they are not shown.

An additional compartment can be used to describe the responsibility of the class. This compartment appears under the services compartment and can be omitted. The responsibility of the class is what the



class will perform. It is displayed as contractual statements. These responsibilities are transformed into services and attributes. Attributes are transformed into datatypes and data structures and services are transformed into methods. This compartment can be labeled "responsibilities". The responsibilities of the student\_schedule class can be stated as: "returns the schedule for a student for any day of the week, the student number, the year, and term of the stored schedule." The responsibilities of the class are displayed as text in its compartment where each responsibility is listed as a short statement or paragraph.

The class diagram can show an object, an instance of a class. Like a class, the simplest representation of an object is a rectangle that contains the name of the object underlined. This is called a named instance of a class. A named instance of a class can be shown with or without its class name:

mySchedule

named instance

mySchedule:student\_schedule

named instance with class name

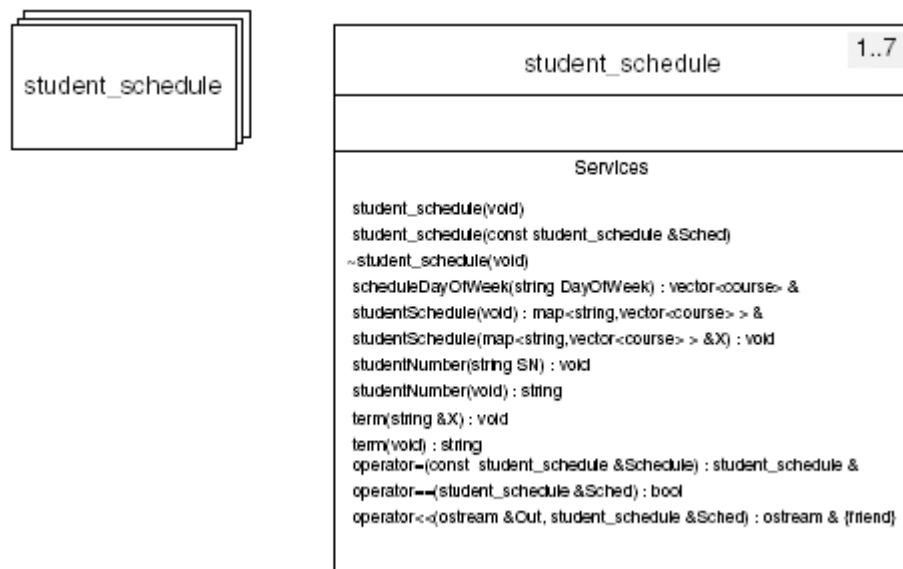
Since the actual name of the object may be known only to the program that declares it, you may want to represent anonymous instances of classes in your system documentation. You can label an object as anonymous in this way:

:student\_schedule

This type of labeling may be convenient where there are several instances of a class in your system. Several instances of a class can be represented in two ways: as objects and as classes.

The number of instances a class may have is called multiplicity. The number of instances of a class can be noted in a class diagram. A class may have zero to an infinite number of instances. A class with zero instances is a pure abstract class. It cannot have any objects explicitly declared of its type. The number of instances may have an upper or lower bound, which may also be expressed in the diagram of a class. [Figure 10-2](#) shows how several instances of a class can be represented in a class diagram as objects or with multiplicity notation.

**Figure 10-2. Multiple instances of a class represented graphically and using multiplicity notation.**



In [Figure 10-2](#), the multiplicity of the student\_schedule class is 1..7, meaning the least number of student schedules in our system is 1 and the most that can exist is 7. Here are more examples of

multiplicity notation and their meaning:

1	One instance
1..n	One to a specified number n
1..*	One to an infinite number
0..1	0 to 1
0..*	0 to an infinite number
*	An infinite number

Of course, an infinite number of instances will be limited by internal memory or external storage.

#### 10.1.1.1 Displaying Specifics about Attributes and Services

The class diagram can specify more details about the attributes and services of the class. The attributes compartment can specify the datatype and/or default value (if there is one) for classes and values of attributes for objects. For example, the datatypes for the attributes of the `student_schedule` class can be displayed:

```
StudentNumber : string
Term : string
StudentSchedule : map <string,vector<course> >
ScheduleIterator : map <string,vector<course> >::iterator
```

For the `mySchedule` object, these attributes can take on values:

```
StudentNumber : string = "102933"
Term : string = "Spring"
```

Methods can be shown with parameters and return type:

```
studentSchedule(&X : map <string,vector<course> >) : void
studentNumber() : string
```

The `studentSchedule()` function accepts the courses of the student. `course` is a class that models a single course. The courses for each day of the week are stored in a vector. The map container maps a string (day of the week) with the vector of courses for that particular day. The `studentSchedule()` function returns void where the `studentNumber()` function returns a string.

The properties of attributes and methods can be displayed in the class diagram. Properties help describe how an attribute or operation can be used. Property labels can be used to describe attributes that are constant or modifiable. There are three properties used to describe attributes: `changeable`, `addOnly`, and `frozen`. [Table 10-1](#) lists these properties with a brief description. There are four properties used to define methods: `isQuery`, `sequential`, `guarded`, and `concurrent`. They are also listed in [Table 10-1](#). `sequential`, `guarded`, and `concurrent` properties are concerned with the concurrency of a method. The `sequential` property describes a concurrent operation where synchronization is the responsibility of the callers of the operation. These operations do not guarantee the integrity of the object. The `guarded`

property describes a concurrent operation where synchronization is already built in. guarded operations mean callers invoke the operation one at a time. The concurrent property describes an operation that permits simultaneous use. The guarded and concurrent operations guarantee the integrity of the object. Guaranteeing the integrity of an object is applicable to operations that change the state of the object.

**Table 10-1. Properties for Attributes and Operations**

**Properties for Attributes**

{changeable}	No restrictions on modifying the values of this type of attribute.
{addOnly}	For attributes with multiplicity > 1, additional values can be added. Once created a value cannot be removed or changed.
{frozen}	Attribute's value cannot be changed once the object has been initialized.

**Properties for Methods**

{isQuery}	Execution of this type of method leaves the state of the object unchanged. This method returns values.
{sequential}	Users of the object must use synchronization to ensure sequential access to this method. Multiple concurrent access to this method jeopardizes the integrity of the object.
{guarded}	Synchronized sequential access to this method is built in the object; integrity of the object is guaranteed.
{concurrent}	Multiple concurrent access is permitted; integrity of the object is guaranteed.

guarded and concurrent properties for methods can be used to reflect the PRAM (Parallel Random-Access Machine) model. If a method is reading and/or writing memory that is accessible to another method that is also reading and/or writing that same memory, then that method can be described as a PRAM algorithm. The properties can be appropriately used. For example:

PRAM Algorithms	Properties
CR (Concurrent Read)	concurrent
CW (Concurrent Write)	concurrent
CRCW (Concurrent Read Concurrent Write)	concurrent

EW (Exclusive Write)	guarded
ER (Exclusive Read)	guarded
EREW (Exclusive Read Exclusive Write)	guarded

The `student_schedule` class can further describe how its attributes and services can be used by using property labels:

attributes

```
StudentNumber : string {frozen}
Term : string {changeable}
StudentSchedule : map <string,vector<course> > {changeable}
```

operations

```
scheduleDayOfWeek(&X : vector<course>, Day : string) :
    void {guarded}
studentNumber() : string {isQuery, concurrent}
```

`StudentNumber` is a string constant. Once an object assigns a value, it cannot be changed. If the `student_schedule` object is used for the same student but for different terms, then `Term` and `StudentSchedule` would be modifiable attributes. The `scheduleDayOfWeek()` operation accepts a vector of courses for a particular day of the week stored in the string `Day`. This operation is guarded. It inserts a student schedule for a particular day of the week into the map object `StudentSchedule`, changing the state of the object. Synchronization is built into the object by using mutexes. The `studentNumber()` operation has two properties: `isQuery` and `concurrent`. It returns the constant `StudentNumber` and is safe for simultaneous access. Calling this method does not change the state of the object thus using the `isQuery` property.

Another important property that can be shown is the visibility of attributes and operations. A visibility property describes who can access the attribute or invoke the operation. This property uses a character or symbol to represent the level of visibility. Visibility maps to the access specifiers of C++:

Access Specifiers	Visibility Symbols
public	(+) Anyone has access.
protected	(#) The class itself and its descendants have access.
private	(-) Only the class itself has access.

The symbol is prepended to the service, method, or attribute name.

#### 10.1.1.2 Ordering the Attributes and Services

It may be best when representing a class with many attributes and operations to organize them within

their compartments. Order helps to identify and navigate through the attributes and operations. The organization can be alphabetical, by access, or by category. Alphabetical order is not helpful in identifying what attributes or operations can be called (if the documentation is targeted to users of the system) or which of them are not defined (if documentation is used in the development process). Ordering by access is very useful. It communicates to the user which attributes and operations are publicly accessible. Knowing which members are protected will assist users who need to extend or specialize the class through inheritance. This can be done by using the visibility symbols, +, -, and #, or by using the C++ access specifiers, protected, public, and private.

There are several ways to categorize the attributes and operations. The minimal standard interface defines categories for operations that in turn define attributes that support these operations. The minimal standard interface is based on the concept that all classes should define certain operations and services in order for a class to be useful. These operations are:

- default constructor
- destructor
- copy constructor
- assignment operations
- equality operations
- input and output operations
- hash operations
- query operations

These can be used as categories to classify the operations of a class. Other categories can be used to help organize attributes and operations:

attributes

static  
const

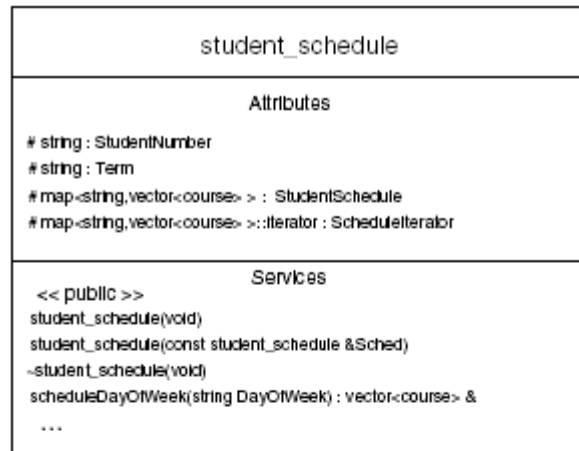
operations

virtual  
pure virtual  
friend

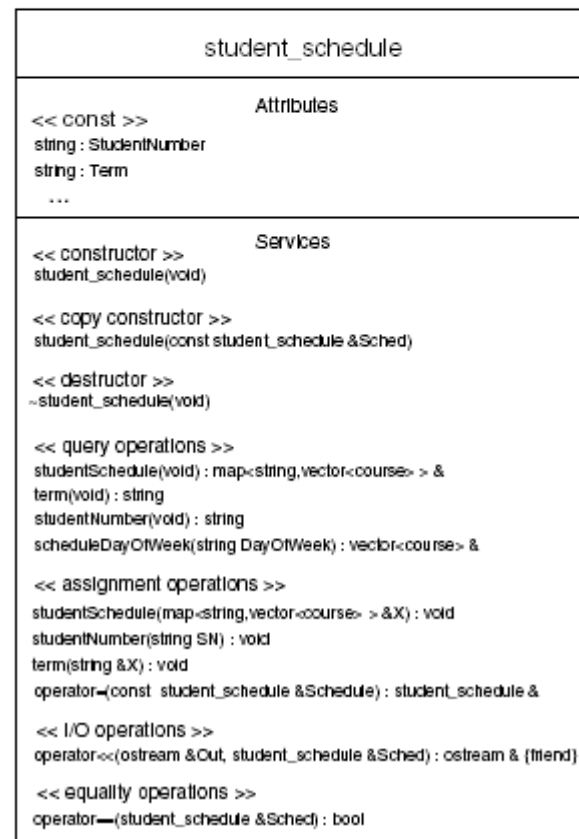
These categories should be used based upon what best describes the services offered by the class. The category name is embraced in left and right double angle brackets, (<<...>>). [Figure 10-3](#) shows the two ways attributes and operations can be organized for the `student_schedule` class: (a) using the visibility symbols, access specifiers, and (b) using categorization based on the minimal standard interface.

Figure 10-3. Two ways attributes and services can be organized in a class diagram.

(a) visibility symbols and access specifiers



(b) categorization based on minimal standard interface



### 10.1.1.3 Template Classes

A template class is a mechanism that allows a type to be a parameter in the definition of the class. The template defines services that manipulate whatever datatype that is passed to it. The parameterized class is created in C++ by using the template keyword:

```
template <class Type > classname {...};
```

Type parameter represents any type passed to the template. Type can be a built-in datatype or a user-defined class. When Type is declared, the template is bound by the element passed to it as the parameterized type. For example, the student\_schedule class has a map container that contains vectors

of course objects for each day of the week. Both the map and the vector are template classes:

```
map <string,vector<course> > StudentSchedule;
```

The map container has string as a key and vector as the value. The vector container contains a user-defined course object. The map container can map any datatype to any other datatype and vector containers can contain any datatype:

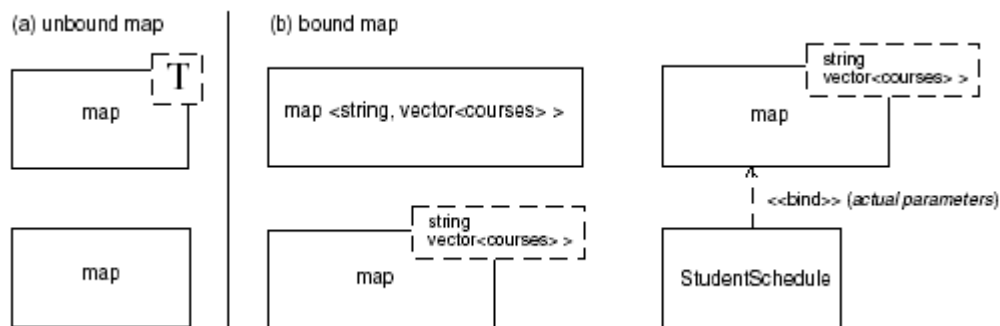
- map <int, vector <string> >                    Maps a number to a vector of strings
- map <int, string> >                            Maps a number to a string
- vector <student\_schedule>                    A vector of student\_schedule objects
- vector <map <int,string> >                    A vector of maps that maps a number to a string

Template classes are also represented as rectangular boxes. The parameterized type is represented as a dashed box displayed in the upper right-hand corner. The template class can be unbound or bound. When representing an unbound template class, the dashed box displays a capital T to represent the unbound parameterized type. There are two ways to represent a bound template class. One approach is to use the class symbol containing the C++ syntax for declaring and binding a template class:

```
vector <string>
```

This is called implicit binding. Another approach uses a dependency stereotype, bind. The stereotype specifies the source instantiating the template class by using the actual parameterized type. This is called explicit binding. The template object is the instantiation of the template class. It has a dependency relationship with the template class. The stereotype specifies the name of the parameter types. Inside the dashed box, datatypes are displayed. The template object can also be considered as a refinement of the template class. Refinement is a general term to indicate a greater level of detail of something that already exists. The stereotype indicator <<bind>> refines the template class by instantiating the parameterized type. [Figure 10-4](#) depicts the ways a template class can be represented, unbound and bound, for a map container.

**Figure 10-4. The ways to represent a bound and unbound template class.**



### 10.1.2 The Relationship between Classes and Objects

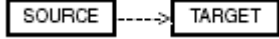
The UML defines three types of relationships between classes:

- dependencies

- generalizations
- associations

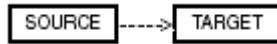
Dependency defines a relationship between two classes. When one class depends on another class, this means a change to the independent class may affect the dependent class. Generalization is a relationship between a general construct and a more specific type of that construct. The general construct is the parent or superclass and the more specific construct is the child or subclass. The child inherits the properties, attributes, and operations of the parent and may define other attributes and operations of its own. The child is derived from the parent and can be used as a substitute for the parent class. A class that has no parent is called the root or base class. Association is a structural relationship that specifies that objects of one type are connected to objects of another type. Associations between objects are bidirectional. For example, if object 1 is associated with object 2, then object 2 is associated with object 1. An association between two elements (classes, etc.) is called a binary association. An association between n elements is called n-ary association.

**Table 10-2. Stereotypes That Can Be Applied to Dependencies**

Dependency	Description
	
stereotype << bind >>	Stipulates that the source instantiates the template target using the actual parameters.
stereotype << friend >>	Stipulates that the source is given visibility into the target.
stereotype << instanceOf >>	Stipulates that the source is an instance of the target; used to define relationships between classes and objects.
stereotype << instantiate >>	Stipulates that the source creates instances of the target; used to define relationships between classes and objects.
stereotype << refine >>	Stipulates that the source is a greater level of detail than the target; used to define relationships between derived and base classes.
stereotype << use >>	Stipulates that the source depends on the public interface of the target.
stereotype << become >>	Stipulates that the target object is the same object as the source, but at a later time in the object's lifetime; target may have different values, states, etc.
stereotype << call >>	Stipulates that the source object invokes the target's method.
stereotype << copy >>	Stipulates that the target object is an exact independent copy of the source object.



## Dependency



## Description

- stereotype << access >> Stipulates that the source package is the given the right to reference the elements of the target package.
- stereotype << extend >> Stipulates that the target use case extends the behavior of the source use case.
- stereotype << include >> Stipulates that the source use case can include the behavior of the target use case at a location named by the source use case.

Dependency, generalization, and association are actually classifications of relationships. There are many types of dependencies, generalizations, and associations that exist and can be defined. Each relationship classification has its own symbol of representation. That symbol is a solid or dashed line segment between the elements and may be accompanied with some type of arrowhead. To further define that relationship to a specific type, stereotypes or adornments are used in conjunction with the line segment. Stereotypes are labels used to further describe a UML element. It is rendered as a name enclosed by guillemets and placed above or next to the element. For example:

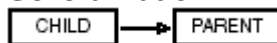
<<bind>>

was placed next to the arrow, which depicts dependency when describing the template object in [Figure 10-4](#). Adornments are textual or graphical items added to an element's basic representation and are used to document details about that element's specifications. For example, an association is depicted as a solid line between elements. Aggregation is a type of association that expresses a "whole-part" relationship. To depict aggregation, a hollow diamond adorns the solid line at the whole end.

Dependency is rendered as a dashed directed line (has a arrow) pointing to the construct being depended on. Use a dependency relationship when one construct uses another. Generalization is rendered as a solid directed line with a large open arrowhead pointing to the parent or superclass. Use a generalization relationship when one construct is derived from another construct. Association is rendered as a solid line connecting the same or different constructs. Use an association relationship when one construct is structurally related to another. [Table 10-2](#) lists some of the stereotypes and constraints that can be applied to dependencies. These stereotypes are used to show dependencies between classes, interactive objects, states, and packages. [Tables 10-3](#) and [10-4](#) list the stereotypes and constraints that can be applied to generalizations and associations. If any of the stereotypes use graphical adornments, they are shown.

**Table 10-3. Stereotypes and Constraints That Can Be Applied to Generalizations**

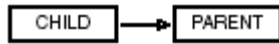
## Generalization



## Description

- stereotype << implementation >> Stipulates that the child inherits the implementation of the parent but does not make public nor support the parent's interfaces.

## Generalization



## Description

constraint {complete}

Stipulates that all children in the generalization have been named and no more additional children can be derived.

constraint {incomplete}

Stipulates that not all children in the generalization have been named and additional children can be derived.

constraint {disjoint}

Stipulates that the parent's objects may have no more than one of its children as a type.

constraint {overlapping}

Stipulates that the parent's objects may have more than one of its children as a type.

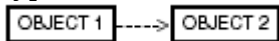
**Table 10-4. Stereotypes, Constraints, and Adornments That Can Be Applied to Associations**

## Association



## Description

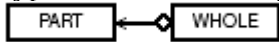
type



navigation

Describes a one-direction association where object 1 is associated with object 2 but object 2 is not associated with object 1.

type



aggregation

Describes a containment (whole-part relationship) where the part is not associated with just one whole for its lifetime.

type



composition

Describes a containment (whole-part relationship) where the part can only be associated with one whole for its lifetime.

constraints {implicit}

Stipulates that the relationship is conceptual.

constraints {ordered}

Stipulates that the objects at one end of the association has an order.

property {changeable}

Describes what can be added, deleted, and changed between two objects.

property {addOnly}

Describes new links that can be added to an object on the opposite end of the association.

## Association



## Description

property {frozen}

Describes a link that once added to an object on the opposite end of the association, cannot be changed or deleted.

Associations have another level of detail that can be applied to a general association or stereotype listed in [Table 10-4](#):

name	An association can have a name that is used to describe the nature of the relationship. A direction triangle can be added to the name to ensure its meaning. The triangle points in the direction the name is intended to be read.
role	A role is the face the class at the near end of the association presents to the class at the other end of the association.
multiplicity	Multiplicity notation can be used to state how many objects may be connected across an association. Multiplicity can be shown at both ends of an association.
navigation	Navigation across an association can be directed where object 1 is associated with object 2 but object 2 is not associated with object 1.

### 10.1.2.1 Interface Classes

An interface class is used to modify the interface of another class or set of classes. The modification makes the class easier to use, more functional, safer, or semantically correct. An example of an interface class are the container adaptors that are part of the Standard Template Library. The adaptors provide a new public interface for the deque, vector, and list containers. [Example 10.1](#) shows the stack class. It is used as an interface class to modify a vector class.

**Example 10.1 Using the stack class as an interface class.**

```
template < class Container >
class stack{
//...
public:
    typedef Container::value_type value_type;
    typedef Container::size_type size_type;
protected:
    Container c;
public:
    bool empty(void) const {return c.empty();}
    size_type size(void) const {return c.size(); }
    value_type& top(void) {return c.back(); }
    const value_type& top const {return c.back(); }
    void push(const value_type& x) {c.push.back(x); }
    void pop(void) {c.pop.back(); }
};
```

The stack is declared by specifying the Container type:

```
stack < vector< T> > Stack;
```

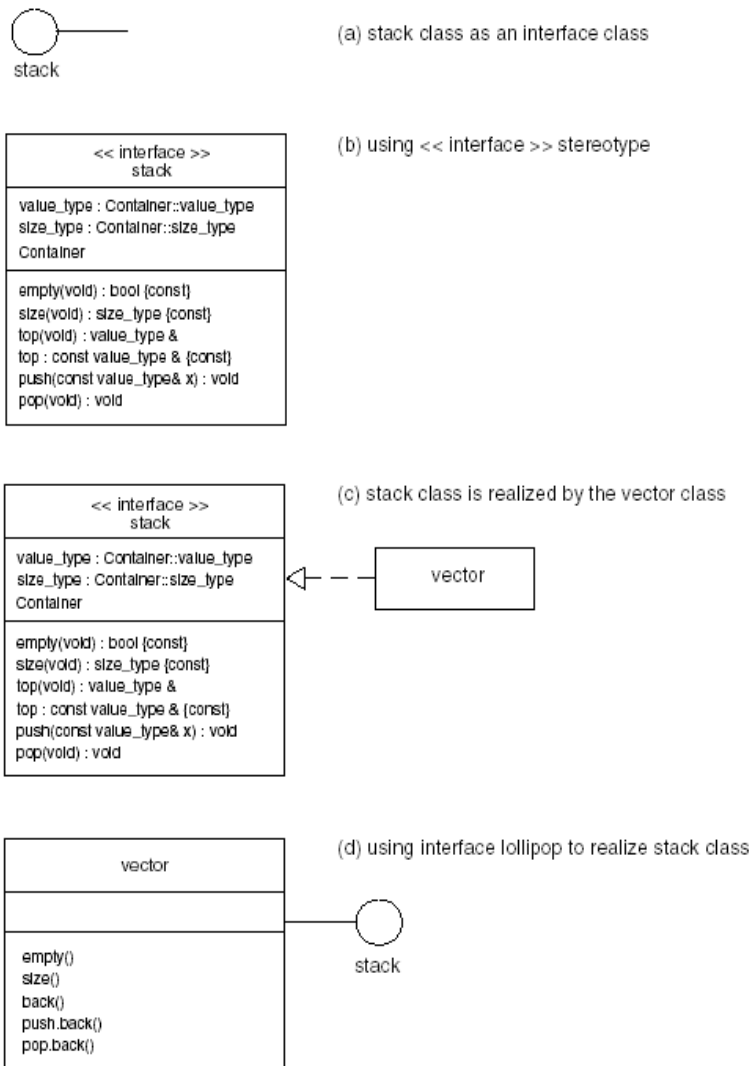
In this case, the Container is a vector but any container that defines these operations:

```
empty()
size()
back()
push.back()
pop.back()
```

can be used as the implementation class for the stack interface class. The stack class supplies the semantically correct interface traditionally accepted for stacks.

There are multiple ways to depict an interface. A circle with the name of the interface class outside the circle is one way to represent an interface class. This is depicted in [Figure 10-5\(a\)](#), showing the stack as an interface class. The class symbol can also be used to show the operations of the stack class, [Figure 10-5\(b\)](#). Here the stereotype indicator <<interface>> is displayed above the name of the class to denote that this is an interface class. The letter I can be prepended to the name of the interface class and all of its operations to further distinguish it from other classes.

**Figure 10-5. Ways to represent an interface class.**



Realization can be used to show the relationship between the stack and the vector class. Realization is a

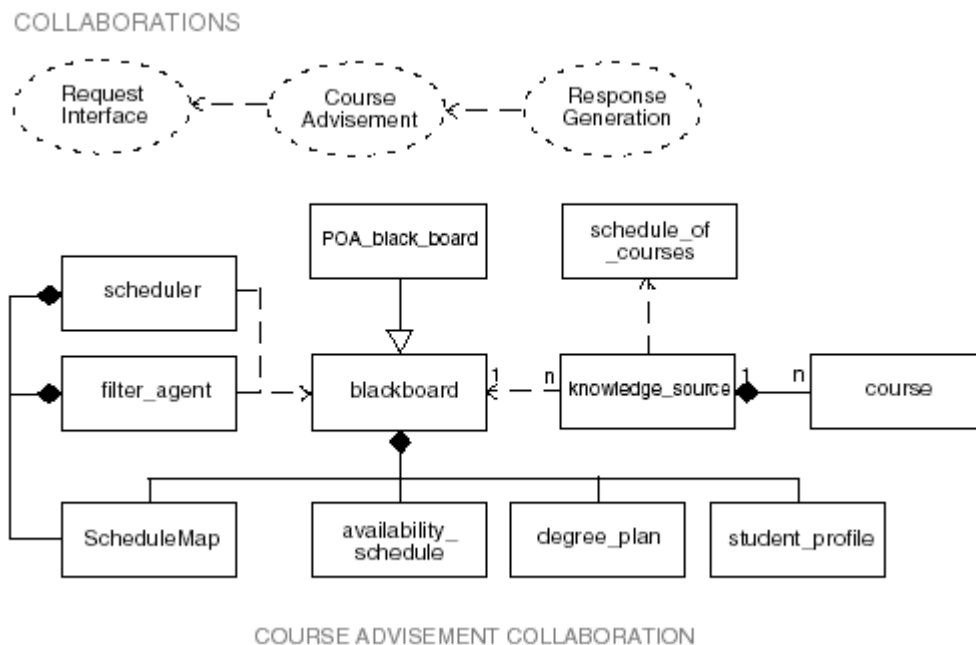
semantic relationship between classes in which one specifies a contract (interface class) and the other class carries it out (implementation class). In our example, the stack class specifies the contract and the vector class carries it out. A realization relationship is depicted as a dashed line between the two classes with a large open arrowhead pointing to the interface class or the class that specifies the contract, which is depicted in [Figure 10-5\(c\)](#). It is read "The stack class is realized by the vector class." The relationship between the interface class and its implementer can also be depicted with the interface lollipop notation, as shown in [Figure 10-5\(d\)](#). The stack class can be the interface to or realized by a vector, list, or deque.

### 10.1.3 The Organization of Interactive Objects

As you can see, classes and interfaces can be used as building blocks to create more complex classes and interfaces. In a distributed or parallel system, there may be many large and complex structures collaborating with other structures, thus creating a society of classes and interfaces working together to accomplish the goals of the system. In the UML, this is called a collaboration. These building blocks can include both the structural and behavioral elements of the system. A particular task requested by a user may involve many objects working together to accomplish that task. Those same objects working with other elements are used to accomplish other tasks. This collection of elements, together with their interactions, form a collaboration. The collaboration has two parts: a structural part, which focuses on the way the collaborating elements are organized and constructed, and a behavioral part, which focuses on the interaction between the elements. This will be discussed in the next section.

A collaboration is depicted as an ellipse with dashed lines containing the name of the collaboration. A collaboration name is unique. It is a noun or short noun phrase based on the vocabulary of the system being modeled. Zooming inside the collaboration ellipse is the structural and behavioral parts of the collaboration. [Figure 10-6](#) shows an example of the structural part of the course adviser system. The structural part of the collaboration consists of any combination of classes and interfaces, components and nodes. In [Figure 10-6](#), a system may contain many collaborations. A single collaboration is unique in the system but the elements of a collaboration are not. The elements of one collaboration may be used in another collaboration using a different organization.

**Figure 10-6. A collaboration diagram for a course adviser system.**



## 10.2 Visualizing Concurrent Behavior

The behavioral view of a system focuses on the dynamic aspects of that system. This view examines how the elements in the system behave as it interacts with other elements of the system. Here is where concurrency will emerge as elements interact with other elements. The diagramming techniques discussed in this section are the ones used to model:

- the lifetime of the behavior of an object
- the behavior of objects that work together for a particular purpose
- flows of control focusing on action or sequence of actions
- synchronization and communication between elements

This section also covers diagramming techniques used to model distributed objects.

### 10.2.1 Collaborating Objects

Collaborating objects are objects involved with each other to perform some specific task. They do not form a permanent relationship. The same objects can be involved with other objects working together to perform other tasks. Collaborating objects can be represented in a collaboration diagram. Collaboration diagrams have a structural part and an interactive part. The structural part has already been discussed. The interaction part is a graph where all of the participating objects are vertices. The connections between the objects are the arcs. The arcs can be adorned with messages passed between the objects, method invocations, and stereotype indicators that express more details about the nature of the connection.

The connection between two objects is a link. A link is a type of association. When two objects are linked, actions can be performed between them. The action may result in a change of the state of one or both objects. These are examples of the types of actions that can take place:

create      An object can be created.

destroy     An object can be destroyed.

call        An operation of an object can be invoked by another object or itself.

return     A value is returned to an object.

send        A signal may be sent to an object.

When any method is invoked, the parameters and the return value can be expressed. Other actions can take place if specified.

These actions can take place if the receiving object is visible to the calling object. Stereotypes can be used to specify why the object is visible:

association    The object is visible because an association exists (very general).

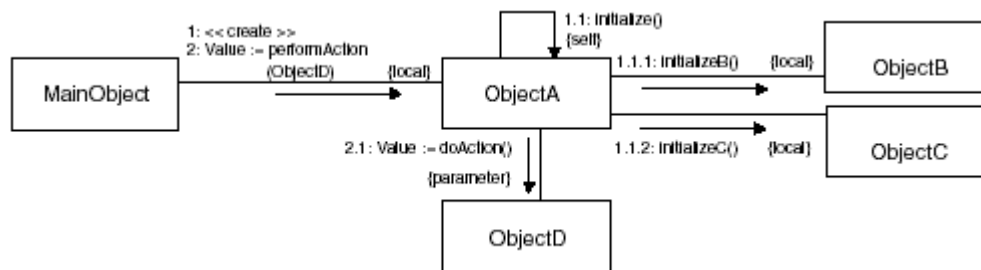
parameter     The object is visible because it is a parameter to the calling object.

- local            The object is visible because it has local scope to the calling object.
- global          The object is visible because it has global scope to the calling object.
- self            The object calls its own method.

Other stereotypes and adornments appropriate for associations can be expressed.

When a method is invoked, this may cause a number of other methods to be invoked by other objects. The sequence in which the operations are performed can be shown by using a sequence number combination and a colon separator prepended to the method. The sequence number combination expresses what sequence the method is associated with and the time order number in which the operation takes place. For example, [Figure 10-7](#) shows a collaboration diagram that uses the sequence numbers.

**Figure 10-7. A collaboration diagram using sequence numbers.**



In [Figure 10-7](#), MainObject performs two operations in sequence:

- 1: <<create>>
- 2: Value := performAction(ObjectF)

In operation 1, MainObject creates ObjectA. ObjectA is local to the MainObject by containment. This initiates the first sequence of operations in a nested flow of control. All operations apart from this sequence use the number 1 followed by the time order number in which the operation takes place. The first operation of sequence 1 is:

1.1: initialize()

ObjectA invokes its own operation. This is expressed by linking the object to itself and by using the {self} stereotype indicator. The ObjectA::initialize() operation also causes the beginning of another sequence of actions:

- 1.1.1: initializeB()
- 1.1.2: initializeC()

in which two other objects local to ObjectA initialize methods are called. The operation:

2: performAction(ObjectD)

is the beginning of another nested sequence. ObjectD is passed to ObjectA. ObjectA invokes ObjectD's operation:

## 2.1: doAction()

ObjectA can invoke this operation because ObjectD is a parameter (passed by MainObject), as the stereotype {parameter} indicates. A value is returned to ObjectA and a value is returned to MainObject. Besides sequence number combinations, these nested flows of control are further enhanced by using a line with a solid arrowhead pointing in the direction of the flow of the sequence.

### 10.2.1.1 Processes and Threads

A process is a unit of work created by the operating system. It has one or more flows of control executing within its address space. Each process has at least one thread, the main thread, but can have many threads executing within its address space. Each thread represents a process's flow of control. Multiple processes can execute concurrently. Threads within the address space of a process can execute concurrently with threads of other processes.

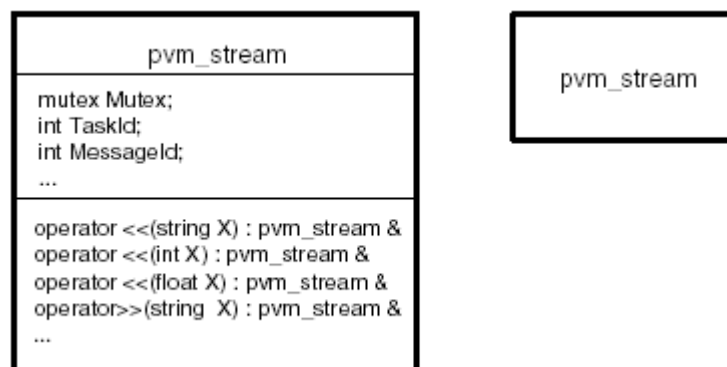
When using the UML, each independent flow of control is considered an active object. An active object is an object that owns a process or thread. Each active object can initiate control activity. An active class is a class whose objects are active. Active classes can be used to model a group of processes or threads that share the same data members and methods. The objects of your system may not have a one-to-one correlation with active objects. As discussed in [Chapters 3 and 4](#), when dividing your program up into processes and threads along object lines, an object's methods may execute in a separate process or execute on separate threads. Therefore, when modeling such an object, it may be represented by several active objects. This relationship between static and active objects can be represented by using an interaction diagram. Your system may have several PVM or MPI tasks or processes. Each of them can be represented directly as an active object.

The UML represents an active object or class the same way a static object is represented, except it has a heavier line tracing the perimeter of the rectangle. Two stereotypes can also be used:

process  
thread

These stereotype indicators can be displayed to show the distinction between the two types of active objects. [Figure 10-8](#) shows a PVM task as an active class and an active object. A collaboration diagram can consist of active objects.

**Figure 10-8. An active object and class.**



### 10.2.1.2 Showing the Multiple Flows of Control and Communication

In a concurrent and distributed system, there will be multiple flows of control. Each flow of control is based on a process or a thread controlling the activity. These processes and threads may be executing



on a single computer system with multiple processors or the processes may be distributed among several different computers. An active object or class is used to represent each flow of control. When the active object is created, an independent flow of control is initiated. When the active object is destroyed, the flow of control is terminated. Modeling the multiple flows of control in your system will help in the management, synchronization, and communication between them.

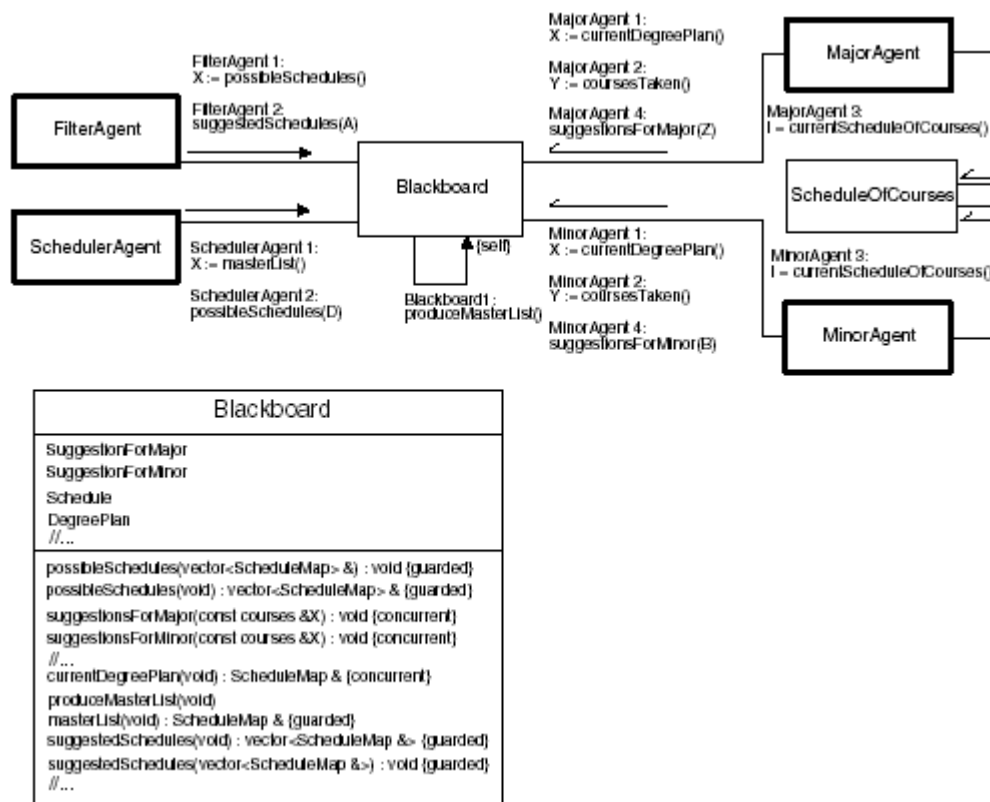
In a collaboration diagram, sequence numbers and solid arrows are used to identify flows of control. In a collaboration diagram that consists of active objects in a concurrent system, the name of the active object is prepended to the sequence numbers of the operations performed by the active object. Active objects can invoke methods in other objects and suspend execution until the function returns or can continue to execute. Arrows are used not to just show the direction of the flow of control but the nature of it. A solid arrowhead is used to represent a synchronous call and a half-stick arrowhead is used to represent an asynchronous call. Since more than one active object can invoke the operation of a single object, the method properties:

- sequential
- guarded
- concurrent

can be used to describe the synchronization property of that method.

[Figure 10-9](#) shows a collaboration of several active objects. In this diagram, these objects are working together to produce a student schedule. The blackboard object is used to record and coordinate the preliminary work and resultant schedule produced by the active object problem solvers, in this case called agents:

**Figure 10-9. A collaboration diagram of static and active objects in the course adviser system.**



MajorAgent                      Produces a list of major courses available.

MinorAgent	Produces a list of minor courses available.
FilterAgent	Filters the list of courses and produces a list of possible courses.
ScheduleAgent	Produces several schedules based on the list of possible courses.

The `schedule_of_courses` object contains all the courses available.

The blackboard and `schedule_of_courses` objects are accessed concurrently by several agents. Both objects are visible to all the agents in this collaboration. The MajorAgent, MinorAgent, FilterAgent, and ScheduleAgent invoke methods of the blackboard object. MajorAgent and MinorAgent invoke methods of the `schedule_of_courses` object. MajorAgent and MinorAgent have a similar sequence of calls to the blackboard and `schedule_of_courses` objects:

MajorAgent1:currentDegreePlan()	MinorAgent1:currentDegreePlan()
MajorAgent2:coursesTaken()	MinorAgent2:coursesTaken()
MajorAgent3:scheduleOfCourses()	MinorAgent3:scheduleOfCourses()
MajorAgent4:suggestionsForMajor()	MinorAgent4:suggestionsForMinor()

As you can see, the name of the active object that invokes these operations are prepended to the sequence number. Both objects are concurrently invoking blackboard and `schedule_of_courses` operations. All these operations have concurrent synchronization and are safe to call simultaneously. `masterList()` and `possibleCourses()` have a guarded property. The objects supplying these courses may be writing them as these objects are attempting to read them. They are guarded by only allowing sequential access (EREW).

## 10.2.2 Message Sequences between Objects

Where a collaboration diagram focuses on the structural organization and interaction of objects working together to perform a task, operation, or realize a use case, a sequence diagram focuses on the time ordering of method invocation or procedures involved in a particular task, operation, or use case. In a sequence diagram, the name of each object or construct involved is displayed in its own rectangular box. The boxes are placed at the top along the x-axis of the diagram. You should only include the major players involved and the most important function calls because the diagram can quickly become too complicated. The objects are ordered from left to right starting from the object or procedure that initiates the action to the most subordinate objects or procedures. The calls are placed along the y-axis from top to bottom in time order. Vertical lines are placed under each box representing the lifeline of the object. Solid arrowhead lines are drawn from the lifeline of one object to the lifeline of another representing a function call or method invocation from the caller to the receiver. Stick arrowhead lines are drawn from the receiver back to the caller representing a return from a function or method. Each function call is labeled at the minimum with the function or method name. The arguments and control information, like the condition in which the method is invoked, can also be displayed. For example:

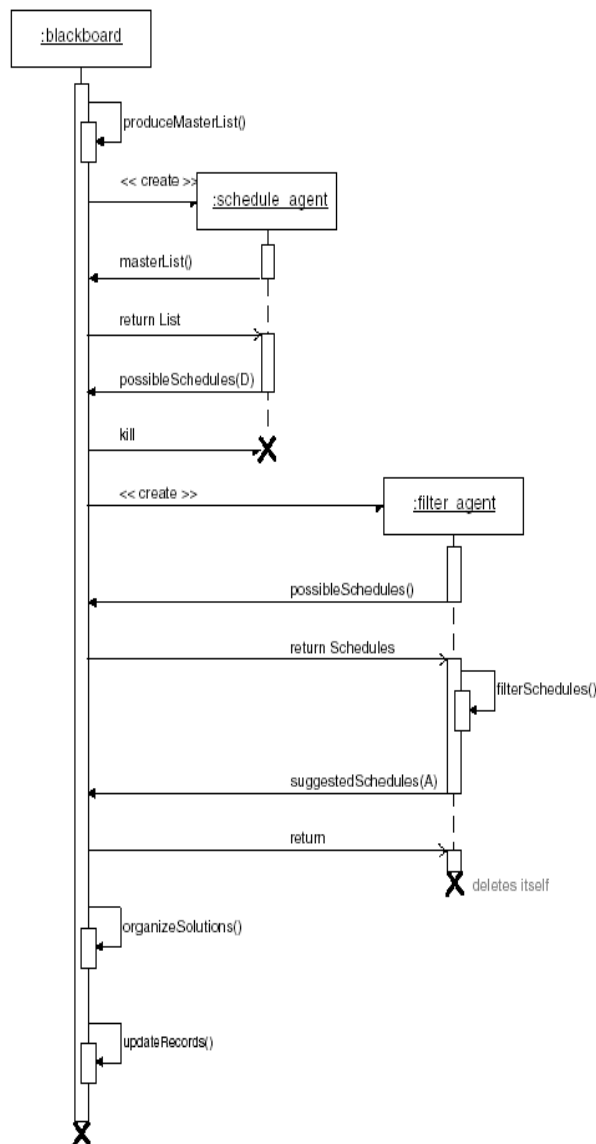
```
[list != empty]
getResults()
```

The function or method will not be performed unless the condition is true. Methods that are to be invoked several times on an object, like reading values from a structure, are preceded by an iteration marker (\*).

[Figure 10-10](#) shows a sequence diagram of some of the objects involved in the course adviser system. Only some of the objects are shown to avoid a complicated diagram. When using the sequence diagram

for concurrent objects or procedures, activation symbols are used. An activation symbol is a rectangle that appears on the object's lifeline. This indicates the object or procedure is active. These are used when an object makes a call to another object or procedure and does not block. This shows that the object or procedure is continuing to execute or be active. In [Figure 10-10](#), the blackboard object is always active. It spawns a schedule\_agent object and does not block. The schedule\_agent calls blackboard.masterList() and waits for the method to return the list of courses. A return arrow is used to indicate the method has returned. The schedule\_agent then calls one of its own methods createSchedules(). To indicate an object has called one of its own methods, a self-delegation symbol is used. This is a combination of an activation symbol and a call arrow. An activation symbol is overlapped on the existing activation symbol. A line proceeds from the original activation symbol with an arrow pointing to the added activation symbol. Once schedule\_agent posts its results by calling blackboard.possibleSchedules(), the blackboard object kills it. This is indicated with the large X at the end of its lifeline. A call arrow from the blackboard object points to this X, indicating it has killed the object. The blackboard object spawns a filter\_agent object and does not block. The filter\_agent calls blackboard.possibleSchedules() and waits for the method to return the schedules. The filter\_agent then calls one of its own methods filterCourses(). Once filter\_agent posts its results, it deletes itself. The blackboard object calls its own organizeSolution() and updateRecords() then deletes itself.

**Figure 10-10. A sequence diagram of some of the objects involved in the course adviser system.**



### 10.2.3 The Activities of Objects

The UML can be used to model the activities performed by objects involved in a specific operation or use case. This is called an activity diagram. It is a flowchart showing the sequential and concurrent actions or activities involved in a specific task, step-by-step. The arrows trace the flow of control for the activities represented in the diagram. Collaboration diagrams emphasize the flow of control from object to object, sequence diagrams emphasize the flow of control in time order, and the activity diagram emphasizes the flow of control from one action or activity to another. The actions or activities change the state of the object or returns a value. The containment of the action or activity is called an action or activity state. They represent the state of the object at a particular instant in the flow of control.

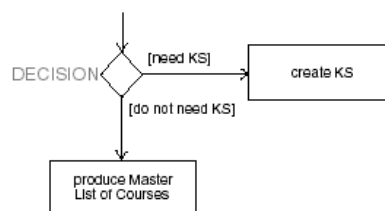
Actions and activities differ. Actions cannot logically be decomposed or interrupted by other actions or events. Examples of actions are creating or destroying an object, invoking an object's method, or calling a function in a procedure. An activity can be decomposed into other activities or even another activity diagram. An example of an activity is a program, a use case, or a procedure. Activities can be interrupted by an event or other activities or actions.

An activity diagram is a graph in which the nodes are actions or activities and the arcs are triggerless transitions. Triggerless transitions require no event to cause the transition to occur. The transition occurs when the previous action or activity has completed. The diagram comprises decision branches, starts, stops, and synchronization bars that join or fork several actions or activities. Both action and activity states are represented the same way. To represent an action or activity state, the UML uses the standard flowchart symbol used to show the enter and exit point of the flowchart. This symbol is used regardless of the type of action or activity occurring. We prefer to use the standard flowchart symbols that distinguish input/output actions (parallelogram) from processing or transformation actions (rectangle). The description of the action or activity as a function call, expression, phrase, use case, or program name is displayed in the action symbol used. An activity state may in addition show the entry and/or exit action. The entry action is the action that takes place when the activity state is entered. The exit action is the action that takes place just before exiting the activity state. They are the first and last actions to be executed in the activity state, respectively.

Once an action has completed, a transition occurs in which the next action takes place immediately. The transition is represented as a directed line from one state with a stick arrow pointing to the next state. A transition pointing to a state is inbound and a transition leading from a state is outbound. Before the outbound transition occurs, the exit action, if it exists, executes. After an inbound transition, the entry action for the state, if it exists, executes. The start of the flow of control is represented as a large solid dot. The first transition leads from the solid dot to the first state in the diagram. The stopping point or stop state of the activity diagram is represented as a large solid dot inside a circle.

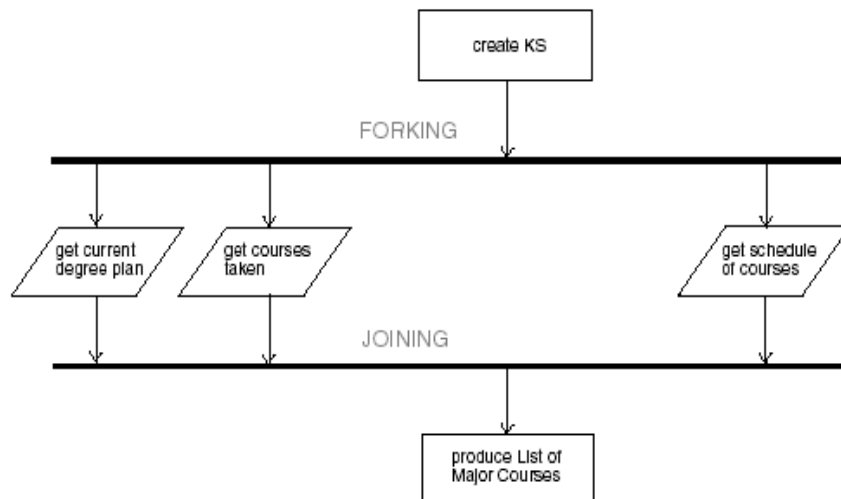
Activity diagrams, like flowcharts, have a decision symbol. The decision symbol is a diamond with one inbound transition and two or more outbound transitions. The outbound transitions are guarded conditions that determine the path of the flow of control. The guarded condition is a simple boolean expression. All of the outbound transitions should cover all of the possible paths from the branch. [Figure 10-11](#) shows the decision symbol used in determining whether a knowledge source should be constructed.

**Figure 10-11. The decision symbol used in determining whether a knowledge source should be constructed.**



You may find that there exists more than one flow of a sequence of actions or activities occurring concurrently after an action or activity has completed. Unlike a flowchart, the UML defines a symbol that can be used to represent the instant where multiple flows of control occur concurrently. A synchronization bar is used to show where a single path branches off or forks into parallel paths and where parallel paths join. It is a thick horizontal line in which there can be multiple outbound transitions (forking) or multiple inbound transitions (joining). Each transition represents a different path. Outbound transitions from a synchronization bar signify an action or activity state has caused multiple flows of control to occur. Inbound transitions into a synchronization bar signify the multiple flows of control need to be synchronized. A synchronization bar is used to show the paths are waiting for all paths to meet and join into a single flow or path. [Figure 10-12](#) shows an example of forking and joining.

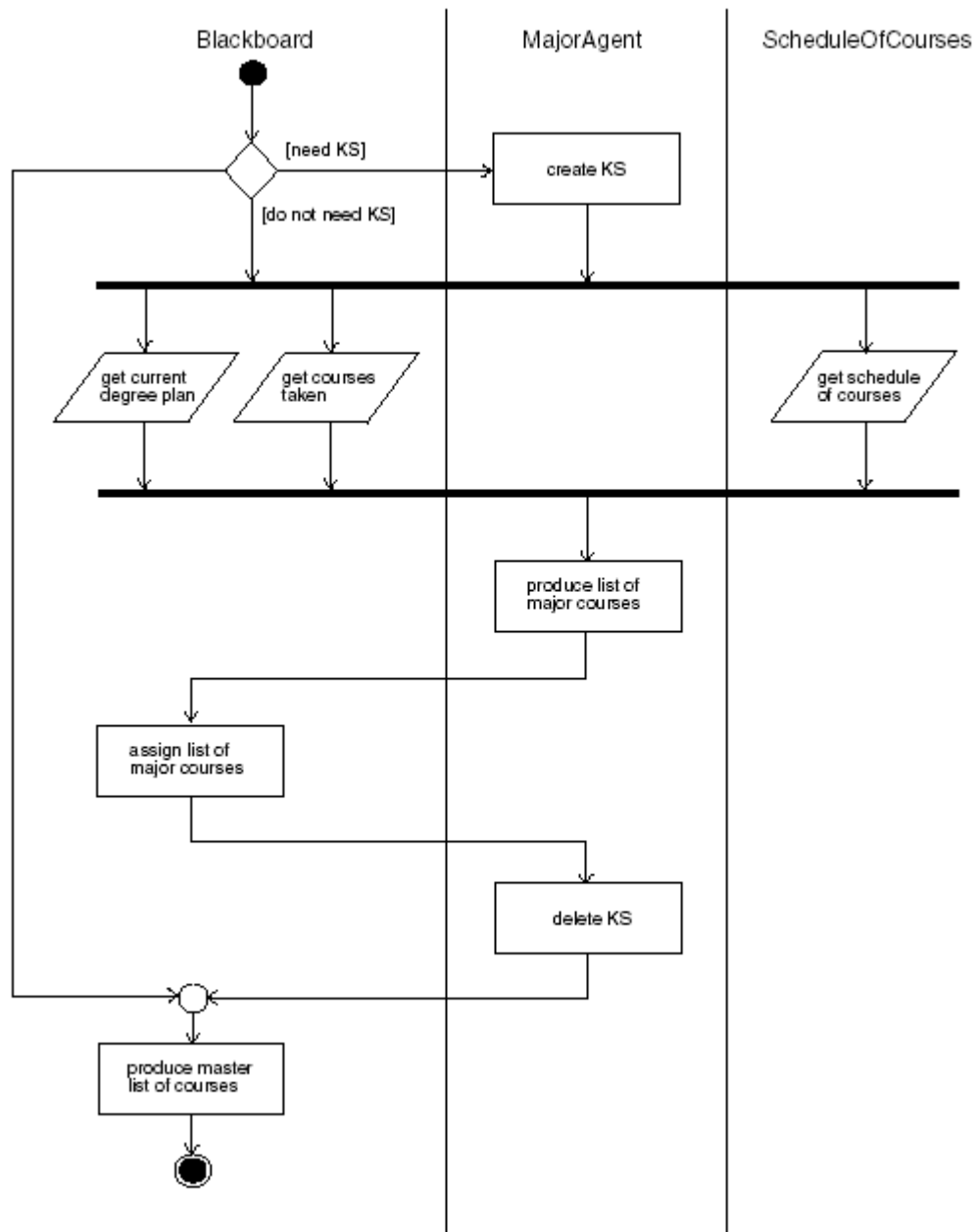
**Figure 10-12. An example of forking and joining from or to the synchronization bar.**



In [Figure 10-12](#), creating MajorAgent invokes its constructor, which forks three flows of control. After these three actions have completed, they are joined again into a single flow of control in which the action "produce list of major courses" is executed.

The diagram can be divided into separate sections called swimlanes. In each swimlane, the actions or activities of a particular object, component, or use case occurs. Swimlanes are vertical lines that partition the diagram into sections. A swimlane for a particular object, component, or use case specifies the focus of activities. An action or activity can only occur in a single swimlane. Transitions and synchronization bars can cross one or more swimlanes. Actions or activities in the same lane or different lanes but at the same level are concurrent. [Figure 10-13](#) shows the activity diagram with swimlanes.

Figure 10-13. An activity diagram with swimlanes showing a sequence of actions in the course advisor system.



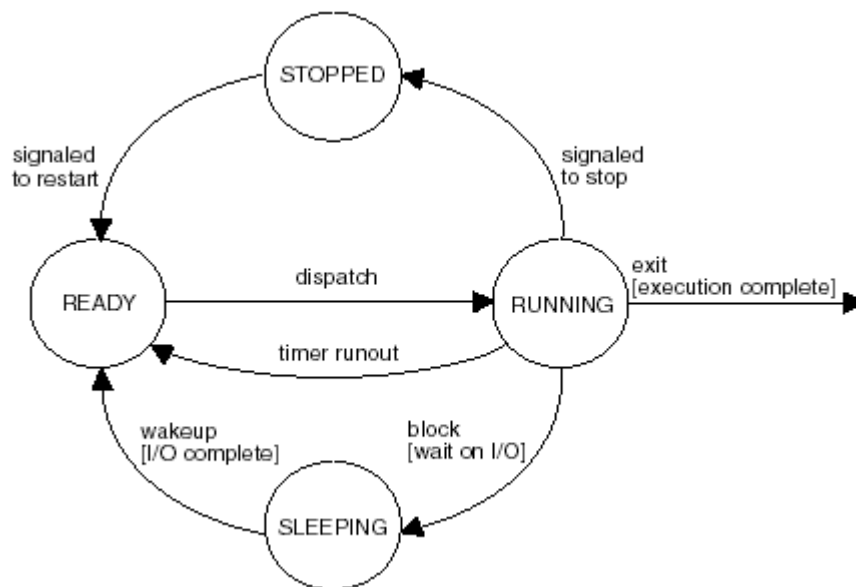
The purpose of this activity diagram is to model the sequence of actions involved in a blackboard object producing the master list for our course adviser system. In [Figure 10-13](#), the blackboard object first makes the decision whether the MajorAgent object should be constructed. If so, the constructor for MajorAgent is invoked. This causes a fork of three flows of control. Two of the actions are executed by the blackboard object, "get current degree plan" and "get courses taken," and one action is executed by the ScheduleofCourses object, "get schedule of courses." These are all input actions, as the symbol represents. The multiple paths are joined again and MajorAgent performs an action "produce list of major courses." The blackboard performs an action "receive list of major courses" followed by the deletion of the MajorAgent object. The blackboard object "produces master list of courses," then the activities stop.

## 10.2.4 State Machines

State machines depict the behavior of a single construct specifying the sequence of transformations during its lifetime as it responds to internal and external events. The single construct can be a system, a use case, or an object. State machines are used to model the behavior of a single entity. An entity can respond to events such as procedures, functions, operations, and signals. An entity can also respond to elapses in time. Whenever an event takes place, the entity responds by performing some activity or taking some action resulting in a change of the state of the entity or the production of some artifact. The action or activity performed will depend upon the current state of the entity. A state is a condition the entity is in during its lifetime as a result of performing some action or responding to some event.

A state machine can be represented in a table or directed graph called a state diagram. [Figure 10-14](#) shows a UML state diagram for the state machine of a process. [Figure 10-14](#) shows the states some process progresses go through while it is active in the system. The process can have four states: ready, running, sleeping, and stopped. There are eight events that cause the four states of the process. Three of the events only occur if a condition is met. The block event occurs only if the process requests I/O or it is waiting for an event to occur. If the block event occurs, it triggers the process to transform from a running state to a sleeping state. The wakeup event occurs only if the event takes place or the I/O has been completed. If the wakeup event occurs, it triggers the process to transform from a sleeping state (source state) to a ready state (target state). The exit event occurs only if the process has executed all its instructions. If the exit event occurs, it triggers the process to transform from a running state to a sleeping state. The remaining events are external events and not under the control of the process. They occur for some external reason triggering the process to transform from a source to a target state.

**Figure 10-14. State diagram for processes.**



The state diagrams are used to model the dynamic aspects of an object, use case, or system. The sequence, activity, and interactive collaboration diagrams and now the state diagram are used to model the behavior of the system or object when it is active. Structural collaboration and class diagrams are used to model the structural organization of an object or system. State diagrams are good to use to describe the behavior of an object regardless of the use case. They should not be used to describe the behavior of several interacting or collaborating objects. They should be used to describe the behavior of an object, system, or use case that goes through a number of transformations and more than one event may cause a single transformation to occur. These are constructs that are very reactive to internal and external events.

In the state diagram, the nodes are states and the arcs are transitions. The states are represented as rounded-corner rectangles in which the name of the state is displayed. The transitions are lines connecting the source and target states with a stick arrow pointing to the target state. There are initial and final states. The initial state is the default starting point for the state machine. It is represented as a solid black dot with a transition to the first state of the state machine. The final state is the ending state of the state machine, indicating it has completed or the system, use case, or object has reached the end of its lifeline. It is represented as a solid dot embedded in a circle.

**Table 10-5. Parts of a State**

**Parts of a State**

Parts of a State	Description
Name	The unique name of the state that distinguishes it from other states; a state may have no name.
Entry / exit actions	Actions executed when entering the state (entry state) or executed when exiting the state (exit action).
Substates	A nested state; the substates are the disjointed states that can be activated sequentially or concurrently. The composite or superstate is the state that contains the substates.
Internal transitions	Transitions that occur within the state that are handled without causing a change in the state.
Self-transitions	Transitions that occur within the state that are handled without causing a change in the state but causes the exit then the entry actions to execute.
Deferred events	A list of events that occurs while the object is in that state but is queued and handled when the object is in another state.

A state has several parts. [Table 10-5](#) lists the parts of a state. A state can be represented simply by displaying the name of the state at the center of the state symbol. If other actions are to be shown inside the state symbol, the name of the state should appear at the top in a separate compartment. The actions and activities are listed below this compartment and are displayed in this format:

label [Guard] / action or activity

For example:

do / validate(data)

The do is the label used for an activity to be performed while the object is in this state. The validate(data) function is called with data as the argument. If an action or activity is a call to a function or method, the arguments can be displayed.

The Guard is an expression that evaluates to true or false. If a condition evaluates to true, the action or activity takes place. For example:



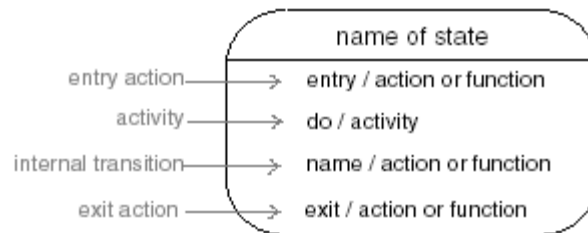
exit [data valid] / send(data)

The exit action send(data) is guarded. The expression data valid is evaluated to be true or false. Upon exiting the state, if the expression is true, then the send(data) function is called. The Guard is always optional.

Transitions occur when an event takes place. This causes the object, system, or use case to transform from one state to another. There are two transitions that can occur that does not cause a change in the state of the object, system, or use case: self-transition and internal transition.

With a self-transition, when a particular event occurs, this triggers the object to leave the current state. When exiting, it performs the exit action (if any), then performs whatever action is associated with the self-transition (if any). The object reenters the state and the entry action (if any) is performed. With an internal transition, the object does not leave the state and therefore no entry or exit actions are performed. [Figure 10-15](#) shows the general structure of a state with exit and entry actions, do activity along with internal and self-transitions. A self-transition is represented as a directed line that points back to the same state.

**Figure 10-15. The general structure of a state with an exit action, entry action, do activity, and internal- self-transitions.**



A transition between different states indicates that there is a relationship or path that exists between them. From one state an event can occur or a condition can be met that causes the object to be transformed from one state (source state) to another state (target state). The event triggers the transition of the object. A transition may have several concurrently existing source states. If so, they are joined before the transition occurs. A transition may have several concurrently existing target states in which a fork has occurred. [Table 10-6](#) lists the parts of a transition. A transition is rendered as a directed line from the source state pointing to the target state. The name of the event trigger is displayed next to the transition. Like actions and activities, events for transitions can also be guarded. A transition can be triggerless, meaning no special event occurs that causes the transition to take place. Exiting the source state, the object immediately makes the transition and enters the target state.

**Table 10-6. Parts of a Transition**

**Parts of a Description Transition**

Source state	The original state of the object; when a transition occurs the object leaves the source state.
Target state	The state the objects enter after a transition occurs.
Event trigger	The event that causes the transition to occur. A transition may be triggerless, in

## Parts of a Description Transition

which the transition occurs as soon as the object has completed all activities in the source state.

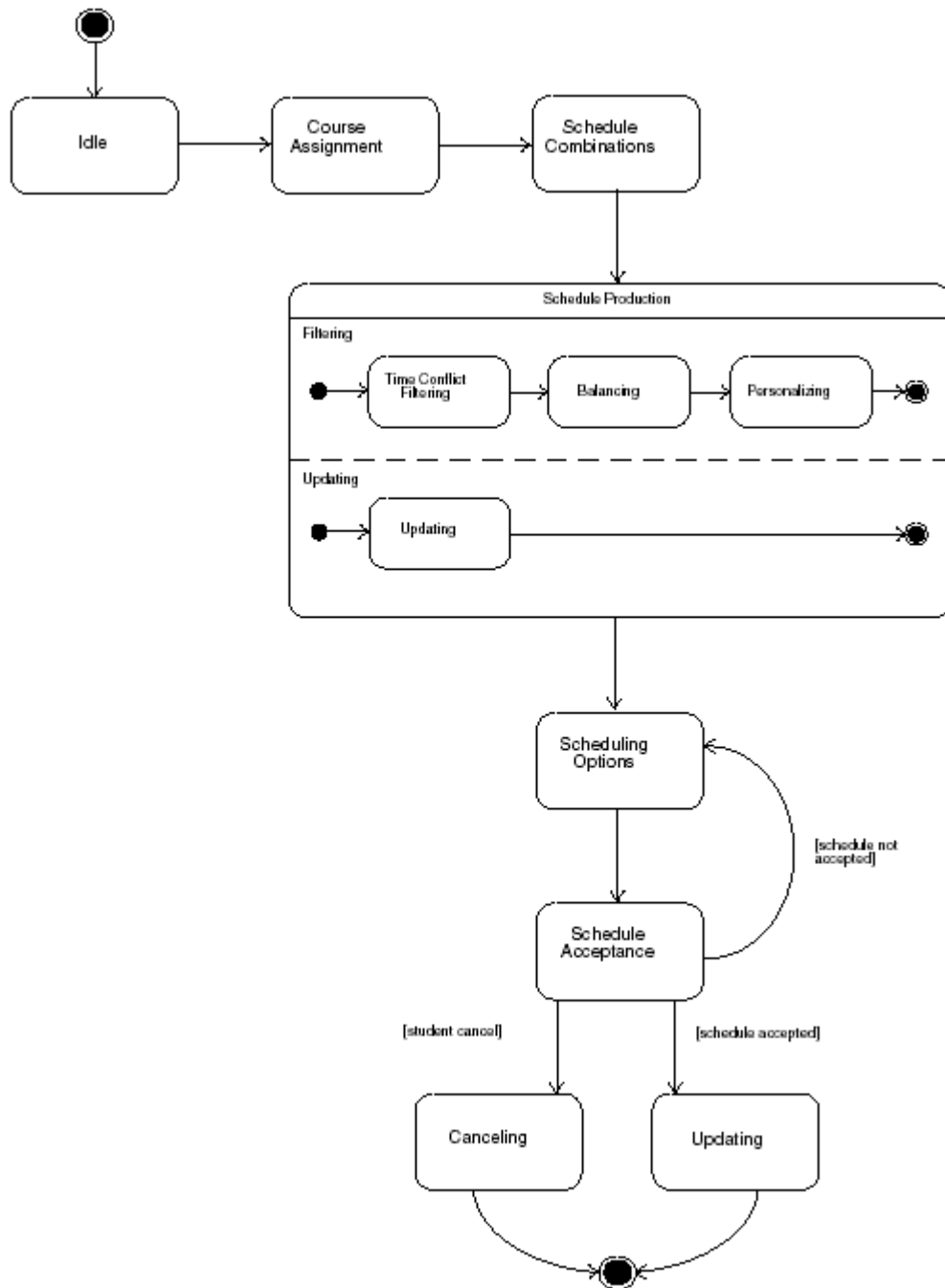
Guard condition A boolean expression associated with an event trigger that when evaluated to True, the transition occurs.

Action An action executed by the object that takes place during a transition; it may be associated with an event trigger and/or guard condition.

### 10.2.4.1 Concurrent Substates

A substate can be used to further simplify the depiction of modeling the behavior of a concurrent system. A substate is a state contained inside another state called a superstate or composite state. This representation means a state can be further broken down into one or more substates. These substates can be sequential or concurrent. With concurrent substates, each state machine represented exists in parallel as different but concurrently existing flows of control. This means the object is engaged in two independent sets of behavior. This is true for our blackboard object. As it is processing each possible schedule, it has to also update its appropriate structures and perform other maintenance. Each substate is contained in a separate compartment. The substates are synchronized and joined before exiting the composite state. When one substate has reached its final state, it waits for the other state to reach its final state, then the substates are joined back into one flow. [Figure 10-16](#) shows a state diagram for the blackboard object that produces a student schedule.

Figure 10-16. State diagram for the blackboard object.



In [Figure 10-16](#), the schedule production is a composite state. It has concurrent substates called filtering and maintenance. Each substate is separated by a dashed line and is represented by its own state machine, each having an initial and final state. In the filtering substate, the object goes through the time conflict filtering, balancing, and then personalizing states. In the maintenance substate, the object goes through one state: updating. When both substates have reached their final states, filtering and maintenance are joined before exiting the composite state schedule production.

### 10.2.5 Distributed Objects

Distributed objects are objects executing on different processors on different machines. A deployment diagram is used to model the view of the system that shows the physical relationships between the software and hardware components in the delivered system. They are used to show how the

components and objects are routed in the distributed system. Components can be executable programs, libraries, or databases. You may want to specify where a particular component or object resides in the system. Determining how to distribute the concurrent components of your system will be difficult. Modeling how the components are distributed will help in managing the configuration, functionality, and throughput of the system.

A deployment diagram consists of nodes and the objects or components that reside on the nodes. A node is a computational unit or a piece of hardware that has some memory and processing capability, whether it be a device, a computer, a mainframe, or a cluster of computers. The nodes are related by dependency. These dependencies represent how the components communicate with each other. The direction of the dependency indicates which component is aware of the other component. Even if there is communication in both directions, one component may not be aware of whom they are communicating with.

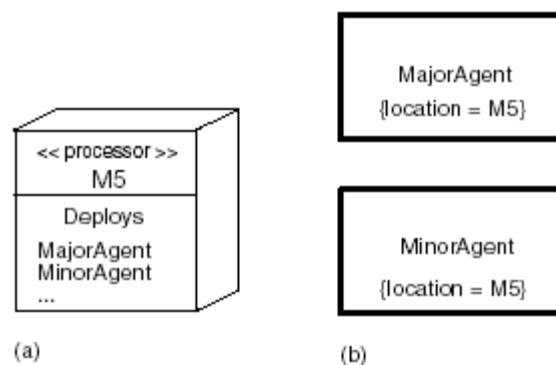
There are two ways to model the location of components or objects in a UML deployment diagram: nesting or tagged value.

They reflect the approach of listing the components that reside on a node in the node symbol or displaying the location of the components in the component symbol. Nodes are a part of a deployment diagram. The node symbol is a cube. The cube can have two separate compartments: one contains the stereotype indicator describing the type of node and the other contains the list of components that reside on that node. The approach uses the component symbol and displays a location tag assigning the name of the node where the component resides. A location tag has this format:

```
{location = name of node}
```

The location tag can be a part of any diagram in which the location of the component is appropriate (e.g., collaboration, object, or activity diagram). [Figure 10-17](#) shows the two approaches of showing the location components in a distributed system. In [Figure 10-17](#), (a) shows the node symbol listing the components that reside on it and (b) shows the active object symbol using the location tag.

**Figure 10-17. Approaches to show the location of a component in a distributed system.**



## 10.3 Visualizing the Whole System

A system is composed of many elements, including subsystems organized into a collaboration to accomplish some purpose. It is an aggregation of constructs joined in some regular interaction. The diagramming techniques discussed in this chapter allow the developer to model a single system from different viewpoints, from different levels, and from different flows of control to assist in the design and development of the system. In this section, we discuss modeling and documenting the system as a whole. This means at the highest level the major components or functional elements can be depicted. The diagramming techniques discussed in this section are the ones used to model the delivered system and the architecture of the system. Although this is the last section in this chapter, modeling and documenting the whole system would be the first level of designing and developing a system.

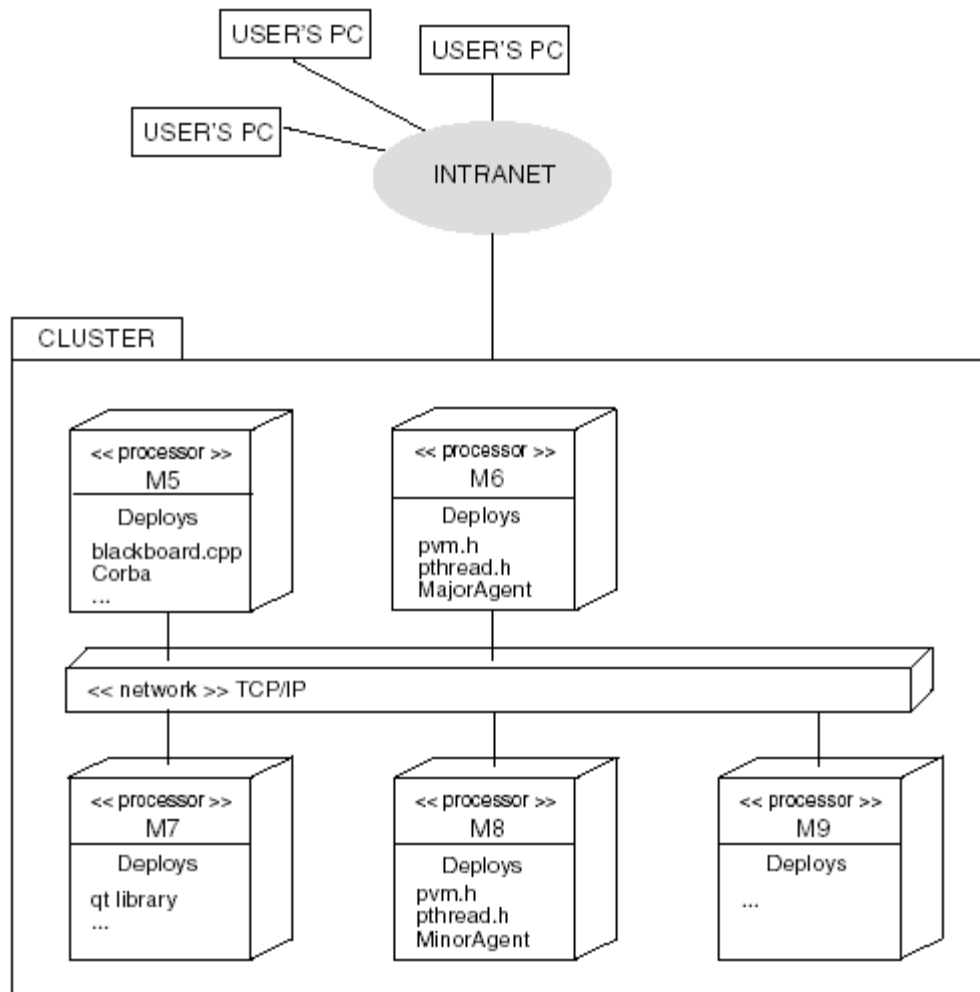
### 10.3.1 Visualizing Deployment of Systems

The deployment of a system is the last step in system development. Deployment is the delivery of the system. When a system is to be deployed, you may want to model the actual physical components of the runtime version of your system. A deployment diagram depicts the configuration of runtime processing elements and the software components that execute on them. The software components are actual executable modules such as active objects (processes), libraries, databases, and so on. A deployment diagram consists of nodes and components. The components used in a deployment diagram are runtime entities. Runtime entities are the physical implementations of logical elements. A class is a logical element that may be implemented as one or several components. A class may be divided into processes or threads. Each process or thread can be a component in a deployment diagram. The components of a class may be executed on different nodes on a single machine (threads/processes) or different machines (processes).

A node is represented by a cube. Nodes are connected by dependencies or associations. Components and nodes can be connected by dependencies as well. As discussed earlier, a node can list its components or a component can be depicted separate from a node showing the relationship between them. A component can be represented as a rectangle with tags on the left side. The name of the component is contained inside the symbol.

Components can be grouped together to create larger chunks such as packages or subsystems. [Figure 10-18](#) shows a deployment diagram. In [Figure 10-18](#), the users connect to the system via intranet. The nodes are the part of a cluster of PCs. They are grouped into a package. The user connects to the cluster as a whole. Each node lists the components that reside on them. The communication between nodes is by means of a network node.

Figure 10-18. A deployment diagram using packages.



### 10.3.2 The Architecture of a System

When modeling and documenting the architecture of a system, the view of the system is at the highest level. Booch, Rumbaugh, and Jacobson define architecture as:

The set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaboration among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization—these elements and their interfaces, their collaborations, and their composition.

Modeling and documenting the architecture will capture the system's logical and physical elements along with the structure and behavior of the system at the highest level.

The architecture of the system is a description of the system from a distinct view that focuses on the structure and organization of the system from that aspect. The views are as follows:

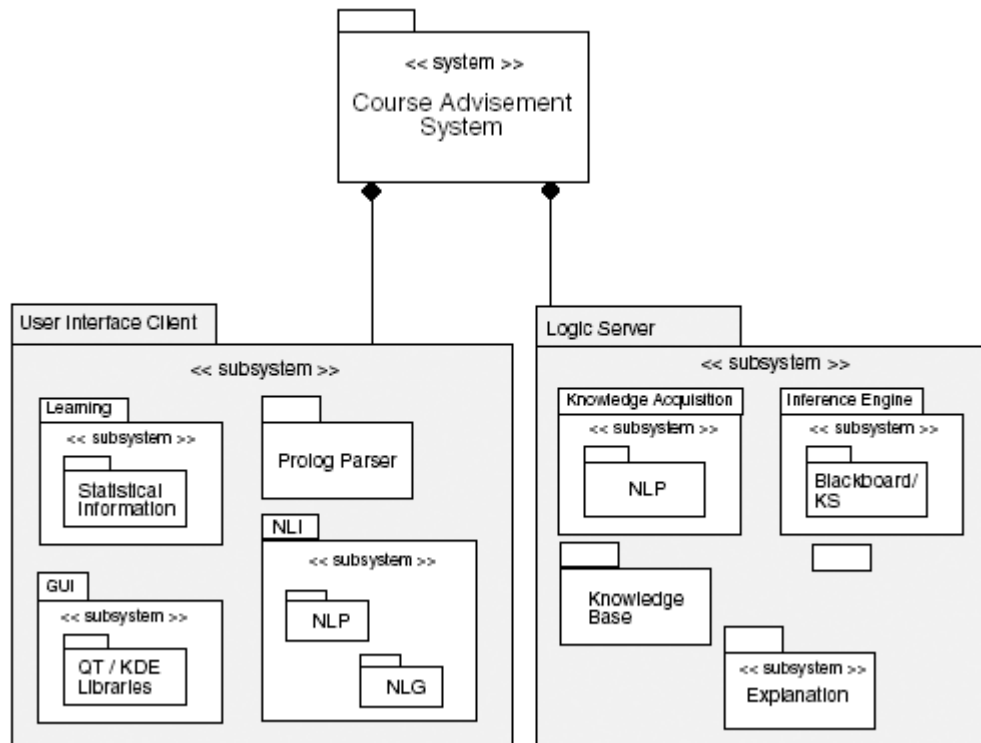
- use case            Describes the behavior of the system presented to end users.
  
- process             Describes the processes and threads used in the system's mechanisms of concurrency and synchronization.

- design Describes the services and functions provided to the end user.
- implementation Describes the components used to create the physical system.
- deployment Describes the software components and the nodes on which they are executing in the delivered system.

As you can see, these views overlap and interact with each other. Use cases can be used in the design view. Processes can show up as components in the implementation view. Software components are used in both implementation and deployment views. When designing the architecture of the system, diagrams that reflect each of these views should be constructed.

A system can be decomposed into subsystems and modules. The subsystems or modules will be further broken down into components, nodes, classes, objects, and interfaces. In the UML, subsystems or modules used at the architectural level of documentation are called packages. A package can be used to organize elements into a group that describes the general purpose of those elements. A package is represented as a rectangle with a tab on the upper left corner. The package symbol contains the name of the package. The packages in the system can be connected by means of composition, aggregation, dependency, and associations relationships. Stereotype indicators can be used to distinguish one type of package from another. [Figure 10-19](#) shows the packages involved in the course adviser system. The system package uses a <<system>> indicator to distinguish it from the User Interface Client and Logic Server subsystems, which use the <<subsystem>> indicator. Because they are subsystems, they are related to the system by aggregation relationship.

**Figure 10-19. Packages used in the course adviser system.**



Packages can contain other packages. If a package contains other packages, then the name of the package is placed in the tab. [Figure 10-19](#) also shows the content of each subsystem.

## Summary

A model of a system is the body of information gathered for the purpose of studying the system. Documentation is a tool used in modeling a system. The UML, Unified Modeling Language, is a graphical notation used to design, visualize, model, and document the artifacts of a software system created by Grady Booch, James Rumbaugh, and Ivar Jacobson. It is the de facto standard for communicating and modeling object-oriented systems. The UML can be used to model concurrent and distributed systems from the structural and behavioral perspectives.

UML diagrams can be used to model to most basic units, the object, to the whole system. An object is the basic unit used in many UML diagrams. Dependency, inheritance, aggregation, and composition are some of the relationships that can exist between objects. Interaction diagrams are used to show the behavior of an object and identify concurrency in the system. Objects can interact with other objects by communicating and invoking methods. Collaboration diagrams depict the interactions between objects working together to perform some particular task. Sequence diagrams are used to represent the interactions between objects in time sequence. Statecharts are used to depict the actions of a single object over its lifetime. Objects that are distributed can be tagged with the location of the node on which it resides.

Deployment diagrams are used to model the delivered system. The basic units of a deployment diagram are nodes and components. A node represents hardware and components are software. Nodes can be depicted to show what objects or components reside on them. When modeling the whole system, the basic unit is a package. A package can be used to represent systems and subsystems. Packages can have relationships with other packages such as composition or some type of association.



# Chapter 11. Designing Components That Support Concurrency

"As we cross the divide to instantiate ourselves into our computational technology, our identity will be based on our evolving mind file. We will be software, not hardware."

—Ray Kurzweil, *The Age of Spiritual Machines*

In this Chapter

- [Taking Advantage of Interface Classes](#)
- [A Closer Look at Object-Oriented Mutual Exclusion and Interface Classes](#)
- [Maintaining the Stream Metaphor](#)
- [User-Defined Classes Designed to Work with PVM Streams](#)
- [Object-Oriented Pipes and fifos as Low-Level Building Blocks](#)
- [Framework Classes Components for Concurrency](#)
- [Summary](#)

As a rule of thumb the requirement for parallelism and concurrency within a piece of software should be discovered and not introduced. Sometimes the goal of speeding up a program is not enough justification to force parallelism into logic that is naturally sequential. The parallelism within a design should be a natural consequence of the requirements of a system. Once concurrency is identified in the system requirements, then architectures and algorithms that support parallelism should be considered. In other cases the need for parallelism will emerge within an existing system that was originally designed with only sequential processing in mind. This is often the case for systems that started as single-user systems and grew into multiuser systems or systems that have evolved functionally far beyond the original specifications. In these systems the requirement for parallelism is after the fact and the system architecture must be augmented to support concurrency. In this book we are concerned with describing techniques for implementing natural parallelism. That is, once we know we need parallelism, how do we do it using C++?

We present an architectural approach to managing parallelism within a program. We take advantage of the C++ support for object-oriented programming and genericity. Particularly C++'s support for inheritance, polymorphism, and templates is used to cleanly implement architectures and components that support concurrency. Object-oriented programming techniques supply support for 10 class types shown in [Table 11-1](#).

**Table 11-1. Types of Object-Oriented Classes**

<b>Types of Classes</b>	<b>Description</b>
Template classes	A parameterized type containing generic code that can use or manipulate any type; an actual type is the parameter for the code body.
Container class	A class used to hold objects in memory or external storage.
Virtual base class	A base class where during multiple inheritance, the class is the indirect and/or direct base of a derived class; only one copy of the class is shared by all the derived classes.
Abstract class	A class that supplies the interface for derived classes that can only be used as a base class; used as the layout for the construction of other classes.
Interface class	A class used to adjust the interface of other classes.
Node class	A class that has added new services or functionality beyond the services inherited from its base class.
Domain class	A class created to simulate some entity within a specific domain; the meaning of the class is relative to the domain.
Aggregate class	A class that contains other classes; has a "whole-part" relationship with other classes.
Concrete class	A complete class whose implementation is defined and instances of the class can be declared; not intended to be a base class and no attempt to create operations of commonality.
Framework class	A class or collection of classes that has a predefined structure and represents a generalized pattern of work.

These class types prove to be especially useful for designs that require concurrency. This is because these class types aid with the building-block approach. We start with primitive components, using them to build synchronization classes. We use the synchronization classes to build concurrency-safe container classes and framework classes. The framework classes are the building blocks for higher level parallel architectures such as multiagent systems and blackboards. At each level the complexity of the parallel and distributed programming is reduced with the help of the various class types in [Table 11-1](#).

We start our discussion with the interface class. An interface or adapter class is used to modify or enhance the interface of another class or set of classes. The interface class may also act as a wrapper around one or more functions that are not members of any particular class. This use of the interface

class allows us to provide an object-oriented interface to software that is not necessarily object-oriented. Furthermore, interface classes allow us to simplify the interfaces of function libraries such as POSIX threads, PVM and MPI. We can either wrap a non-object-oriented function in an object-oriented interface; or we might want to wrap a piece of data, encapsulate it, and give it an object-oriented interface. In addition to simplifying the complexity of some function libraries, the interface classes are used to present a consistent API (Application Programmer Interface) to the developers. For example, C++ programmers that have grown accustomed to the advantages of the iostreams classes tend to think of input and output in terms of object-oriented streams. The learning curve is minimized when new input and output techniques can be presented within the iostream metaphor. For instance, we might present the MPI message passing library as a collection of streams:

```
mpi_stream Stream1;
mpi_stream Stream2;

Stream1 << Message1 << Message2 << Message3;
Stream2 >> Message4;
//...
```

In this way, the programmer can focus on the logic of the program without getting bogged down in the syntax requirements of the MPI library.

## 11.1 Taking Advantage of Interface Classes

It is often advantageous to use encapsulation to hide the details of function libraries and to provide self-contained components that can be reused. Let's take for example the mutex that we discussed in [Chapter 7](#). Recall that a mutex is a special kind of variable used for synchronization. Mutexes are used to provide safe access to a critical section of data or code within a program. There are six basic functions that can be performed on a `pthread_mutex_t` (POSIX Threads Mutex) variable.

### Synopsis

[\[View full width\]](#)

```
#include <pthread.h>

pthread_mutex_destroy(pthread_mutex_t *mutex);
pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t
➤ *attr);
pthread_mutex_lock(pthread_mutex_t *mutex);
pthread_mutex_timedlock(pthread_mutex_t *mutex);
pthread_mutex_trylock(pthread_mutex_t *mutex);
pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Each of these functions take at least a pointer to a `pthread_mutex_t` variable. An interface class can be used to encapsulate access to the `pthread_mutex_t` variable and to simplify the function calls that access the `pthread_mutex_t` variable. In [Example 11.1](#), we can declare a class called `mutex`.

### Example 11.1 Declaration of the mutex class.

```
class mutex{
protected:
    pthread_mutex_t *Mutex;
    pthread_mutexattr_t *Attr;
public:
    mutex(void)
    int lock(void);
    int unlock(void);
    int trylock(void);
    int timedlock(void);
};
```

Once this mutex class is declared, we can use it to define mutex variables. We can declare arrays of these mutexes. We use these variables as members of user-defined classes. By encapsulating the `pthread_mutex_t` variable and its functions, we can take advantage of the object-oriented programming techniques. These mutex variables can now be used as parameter arguments and function return values. And since the functions are now bound to the `pthread_mutex_t` variable, wherever we pass the mutex variable the functions are also available.

The member functions for the class `mutex` are defined by wrapping the calls to the corresponding Pthread routines, for instance:

### Example 11.2 Member functions for the mutex class.

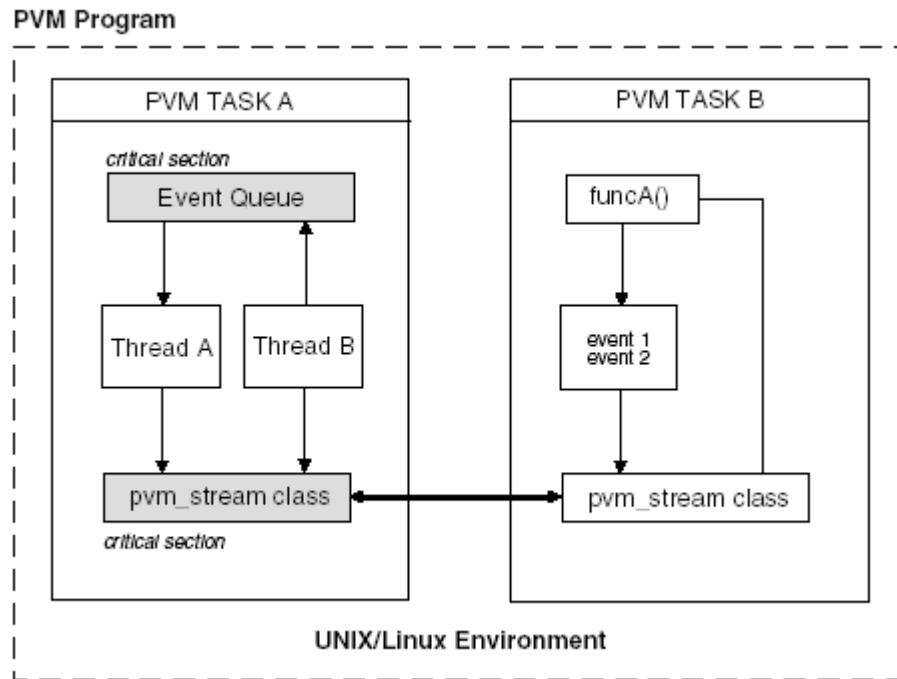
```
mutex::mutex(void)
{
    try{
        int Value;
        Value = pthread_mutexattr_int(Attr);
        //...
        Value = pthread_mutex_init(Mutex,Attr);
        //...
    }
}

int mutex::lock(void)
{
    int ReturnValue;
    ReturnValue = pthread_mutex_lock(Mutex);
    //...
    return(ReturnValue);
}
```

We also protect the `pthread_mutex_t *` and the `pthread_mutexattr_t *` through encapsulation. In other words, when we invoke the `lock()`, `unlock()`, `trylock()`, and so on methods, we don't have to worry about which mutex variable or which attribute variables these functions will be applied to. The availability of information hiding through encapsulation allows the programmer to write safer code. With the free floating versions of Pthread mutex functions, any `pthread_mutex_t` variable may be passed to these functions. Simply passing the wrong mutex to one of these functions can lead to deadlock or indefinite postponement. Encapsulating the `pthread_mutex_t` variable and the `pthread_mutexattr_t` variable within the `mutex` class gives the programmer complete control over which functions have access to those particular variables.

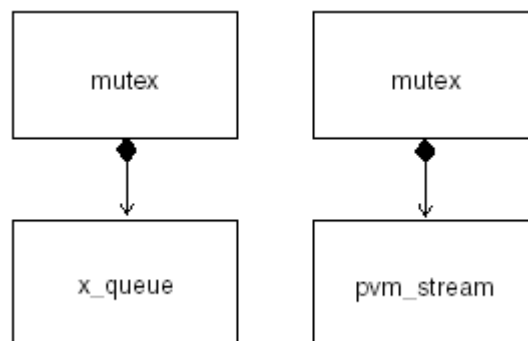
Now we can use an embedded interface class like mutex within other userdefined classes to provide thread-safe classes. Let's say that we wanted to make a thread-safe queue and a thread-safe pvm\_stream class. The queue is used to store incoming events for multiple threads within a program. Some of the threads have the responsibility of sending messages to various PVM tasks. The PVM tasks and the threads are executing concurrently. The multiple threads share a single PVM stream and a single event queue. [Figure 11-1](#) shows the relationship between threads, PVM tasks, event queue, and pvm\_stream.

**Figure 11-1.** The relationship between threads, PVM tasks, event queue, and the pvm\_stream class within a PVM program.



The queue in [Figure 11-1](#) is a critical section because it is shared between multiple executing threads. The pvm\_stream in [Figure 11-1](#) is also a critical section because it is shared between multiple executing threads. If these critical sections are not synchronized and protected, then we can end up with corrupted data in the queue and pvm\_stream. The fact that multiple threads can update either the queue or the pvm\_stream introduces an environment for race conditions. To help manage the race conditions we design our queue and pvm\_stream class with built-in lock and unlock functionality. The lock and unlock functionality is supplied by our mutex class. [Figure 11-2](#) shows the class diagram for our user-defined x\_queue and pvm\_stream classes.

**Figure 11-2.** The class diagram for our userdefined x\_queue class and pvm\_stream class.



Notice that the x\_queue class contains a mutex class. This is a has-a or aggregation relationship

between `x_queue` and `mutex`, that is, `x_queue` has a `mutex` class. Any operation that changes the state of our `x_queue` class can cause a race condition if that operation is not synchronized. Therefore, the operations that add an object to the queue or that remove an object from the queue are candidates for synchronization. [Example 11.3](#) contains the declaration of our `x_queue` class as a template class.

**Example 11.3 Declaration of the `x_queue` class.**

```
template <class T> x_queue class{
protected:
    queue<T> EventQ;
    mutex Mutex;
    //...
public:

    bool enqueue(T Object);
    T dequeue(void);
    //...
};
```

The `enqueue()` method is used to add items to the queue and the `dequeue()` method is used to remove items from the queue. Each of these methods will use the `Mutex` object. The `enqueue()` and `dequeue()` methods are defined in [Example 11.4](#).

**Example 11.4 Definition of `enqueue()` method.**

```
template<class T> bool x_queue<T>::enqueue(T Object)
{
    Mutex.lock();
    EventQ.push(Object);
    Mutex.unlock();
}

template<class T> T x_queue<T>::dequeue(void)
{
    T Object;
    //...
    Mutex.lock();
    Object = EventQ.front();
    EventQ.pop();
    Mutex.unlock();
    //...
    return(Object);
}
```

Now items can be added to and removed from our queue in a multithreaded environment. Thread B in [Figure 11-1](#) adds items to the queue and Thread A in [Figure 11-1](#) removes items. The `mutex` class is an interface class. It wraps the `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pthread_mutex_init()`, and `pthread_mutex_trylock()` functions. The `x_queue` class is also an interface class because it adapts the interface for the built-in `queue<T>` class. First, it changes the `push()` and `pop()` method interfaces to `enqueue()` and `dequeue()`. Furthermore, it wraps the insertion and removal of items with the `Mutex.lock()` and `Mutex.unlock()` methods. So in the first case we use the interface class to encapsulate `pthread_mutex_t*` and `pthread_mutexattr_t*` variables. We also wrap several functions from the Pthread library. In the second case, we use the interface class to adapt the interface of the `queue<T>` class. Another advantage of the `mutex` class is that it can be easily reused in other classes that contain critical sections or critical regions.

In [Figure 11-1](#) the PVM stream is also a critical section because both Thread A and Thread B have access to the stream. A possible race condition exists because Thread A and Thread B may both try to access the stream at the same time. Therefore, we use the mutex class in our user-defined `pvm_stream` class to provide synchronization.

**Example 11.5 Declaration of `pvm_stream` class.**

```
class pvm_stream{
protected:
    mutex Mutex;
    int TaskId;
    int MessageId;
    //...
public:
    pvm_stream & operator <<(string X);
    pvm_stream & operator <<(int X);
    pvm_stream &operator <<(float X);
    pvm_stream &operator >>(string X);
    //...
};
```

As with the `x_queue` class, the `Mutex` object is used with the functions that can change the state of a `pvm_stream` object. For example, we might define one of the `<<` operators as:

**Example 11.6 Definition of the `<<` operator for the `pvm_stream` class.**

```
pvm_stream &pvm_stream::operator<<(string X)
{
    //...
    pvm_pkbyte(const_cast<char *>(X.data()),X.size(),1);
    Mutex.lock();
    pvm_send(TaskId,MessageId);
    Mutex.unlock();
    //...
    return(*this);
}
```

The `pvm_stream` class uses `Mutex` objects to synchronize access to its critical section in the same manner as was done with the `x_queue` class. It's important to note that in both cases the `pthread_mutex` functions are hidden. The programmer does not have to be concerned about their syntax. A simpler `lock()` and `unlock()` interface is used. Furthermore, there is no confusion about which `pthread_mutex_t` \* is being used with the `pthread_mutex` functions. In addition to these advantages, the programmer may declare multiple instances of the `mutex` class without having to call Pthread mutex functions over again. We made reference to the Pthread functions once within the method definitions of the `mutex` class. Now only the methods of the `mutex` class need be called.

## 11.2 A Closer Look at Object-Oriented Mutual Exclusion and Interface Classes

To confront some of the complexity with writing and maintaining programs that require concurrency, we try to streamline and simplify the API to the parallel libraries. Some systems may require the use of the Pthreads library, the MPI library, the standard semaphore, and shared memory functions as part of a single solution. Each of these libraries and functions have their own protocols and syntax. However, often they have similar functionality. We can use interface classes, inheritance, and polymorphism to present a simplified and consistent interface to the programmer. We can also hide the details of library-specific implementation from our applications. If the application only relies on the methods used in our interface classes, then our application is shielded from implementation changes, library updates, and other under-the-hood restructuring. Ultimately the work that you do in providing interface classes to concurrency components and function libraries and function data will allow you to reduce the complexity of parallel programming. Let's take a closer look at how we can approach the design of interface classes that support concurrency.

### 11.2.1 Semi-Fat Interfaces that Support Concurrency

The basic POSIX semaphore is used to synchronize access to a critical section between two or more processes. The basic POSIX thread is used to synchronize access to a critical section between two or more threads. In both cases, there are synchronization variables involved and a number of functions available on the synchronization variables. The MPI library and the PVM library both contain message-passing primitives. Both have the capabilities of spawning tasks. However, the interfaces of these two libraries are different. The application programmer wants to focus on the logic and structure of the program. This is difficult when the semantics of a program are obscured by multiple libraries that happen to perform similar functions but whose syntax and protocols are very different. What is needed is a generalized interface that can be used across libraries.

There are at least a couple of approaches to designing an interface for a family of classes or a collection of classes. The object-oriented approach starts with the general and moves to the specific by means of inheritance. That is, we take the minimal core set of characteristics and attributes that every member of the family of classes should have and, through lineage of inheritance, we specialize those characteristics for each class. In this approach, the interface grows more narrow as you move down the class hierarchy. The second approach is often used in template collections. Template-based approaches start with the specific and move to the general through fat interfaces. The fat interface includes a generalization of all of the characteristics and attributes under discussion (see Stroustrup, 1997). If we were to apply narrow and fat interfaces to our concurrency libraries, the narrow interface approach would take intersection between each library, generalize it, and put it in a base class. The fat interface approach would take the union of the functionality within each library, generalize it, and put in a base class. The set intersection would produce a smaller, less useful class. The set union would produce a large, possibly unwieldy class. For our discussion we are interested in a position somewhere in the middle. We want semi-fat interfaces. We start with a narrow approach and generalize as much as we can within a single class hierarchy. We then use the narrow interface as a basis for a collection of classes that are not related by inheritance but that are related by function. The narrow interface acts as a sort of policy to constrain how fat a semi-fat interface can become. In other words, we don't want a union of every characteristic and attribute under discussion; we only want a union of the things that are logically related to our narrow interface. Let's illustrate this point with a simple design of interface classes for the Pthread mutex, Pthread read-write lock variable, and the POSIX semaphore.

Regardless of the implementation details, the operations of lock, unlock, and trylock are characteristic of synchronization variables. So we make a base class that will act as a pattern for a family of classes. The `synchronization_variable` class is declared in [Example 11.7](#).



**Example 11.7 Declaration of the `synchronization_variable` class.**

```
class synchronization_variable{
protected:
    runtime_error Exception;
    //...
public:
    int virtual lock(void) = 0;
    int virtual unlock(void) = 0;
    int virtual trylock(void) = 0;
    //...
};
```

Notice that the methods of the `synchronization_variable` class are declared virtual and are initialized to 0. This means these methods are pure virtual methods, making the `synchronization_variable` class an abstract class. An object cannot be directly created from any class that has one or more pure virtual functions. In order to use this class a new class must be derived from it and the new class must provide definitions for each of the pure virtual functions. The abstract class acts as a kind of policy that says what functions a derived class must define. It provides an interface blueprint for derived classes. It doesn't dictate how the methods should be implemented, only that the methods must be present and cannot be pure virtuals. We can get hints of the proposed behavior from the names of the methods. The blueprint interface class provides an interface without any implementation. This type of class is used to provide a foundation for future classes. The blueprint class guarantees that the interface will have a certain look (Carroll & Ellis, 1995). The `synchronization_variable` class provides a blueprint interface policy for our family of synchronization variables. We use inheritance to provide implementations for the interface. The Pthread mutex is a good candidate for an interface class, so we define a mutex class derived from `synchronization_variable`:

**Example 11.8 Declaration of a mutex class that inherits the `synchronization_variable` class.**

```
class mutex : public synchronization_variable{
protected:
    pthread_mutex_t *Mutex;
    pthread_mutexattr_t *MutexAttr;
    //...
public:
    int lock(void);
    int unlock(void);
    int trylock(void);
    //...
};
```

The mutex class will provide implementations for each of the pure virtual functions. Once the functions are defined, the policy suggested by the abstract base class has been met. The mutex class is not considered an abstract class, therefore mutex and any of its descendants can be instantiated as objects. Each of the methods of mutex wraps the corresponding Pthread function. For instance:

```
int mutex::trylock(void)
{
    //...
    return(pthread_mutex_trylock(Mutex);
    //...
}
```

provides an interface to the `pthread_mutex_trylock()` function. The `lock()`, `unlock()`, and `trylock()` interface simplifies the Pthread function calls. Our goal is to use encapsulation and inheritance to

eventually define a complete family of mutex classes. The inheritance process is a specialization process. The derived class provides additional attributes or characteristics that distinguish it from its ancestors. Each attribute or characteristic added to the derived class specializes it. Now we can design a specialization of the mutex class through inheritance by adding the notion of a read/write mutex class. Our generic mutex class is designed to protect a critical section from access. Once a thread has locked the mutex, it has access to the critical section the mutex protects. Sometimes this is too extreme. There are times when it is okay to allow multiple threads to access the same data at the same time, so long as none of the threads modify or change the data in any way. That is, there are times that we may want to relax the lock on the critical section and only lock out access to actions that want to modify or change the data and allow access to actions that only read or copy the data. This is called a read lock. The read lock allows concurrent read access to a critical section. The critical section may already be locked by one thread and another thread may also obtain a lock so long as it does not want to modify the data. The critical section may be locked for writing by some thread, and another thread may request a lock for reading the critical section.

The blackboard architecture is a good example of a structure that can take advantage of read mutexes and the stronger, more generic mutex. The blackboard is a common region shared by concurrently executing routines. The blackboard is used to hold solutions to some problem that the group of routines is collaboratively solving. As each routine makes progress toward the solution to the problem, it writes its progress to the blackboard. Each routine also reads the blackboard to see if there are any results generated by the other routines that might be useful. The blackboard structure is a critical section. We really only want one routine at a time to update the blackboard. On the other hand, we can allow any number of routines to simultaneously read the blackboard. Also, if we have multiple routines reading the blackboard, we don't want the blackboard updated until all the routines that are reading are done. The read mutex is an appropriate mutex for this situation because it can lock access to the blackboard and only allow blackboard readers while denying access to blackboard writers. However, the blackboard will need to be updated if a solution to the problem is ever to be achieved. When the blackboard is being updated, we do not want any readers to have access to the blackboard. We want to block the readers until the routine that is updating the blackboard is done. Therefore, we need a write mutex. Only one routine may hold a write mutex at a time. So we distinguish between a mutex that is locked for reading and no writing and a mutex that is locked for writing and no reading. With a read mutex we can have multiple concurrent reads, and with a write mutex we may only have one writer. This is part of the CREW (Concurrent Read Exclusive Write) approach to parallel programming.

## Synopsis

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *,
                        const pthread_rwlockattr_t *);
int pthread_rwlock_destroy(pthread_rwlock_t *);
int pthread_rwlock_rdlock(pthread_rwlock_t *);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *);
int pthread_rwlock_wrlock(pthread_rwlock_t *);
int pthread_rwlock_trywrlock(pthread_rwlock_t *);
int pthread_rwlock_unlock(pthread_rwlock_t *);
int pthread_rwlockattr_init(pthread_rwlockattr_t *);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *);
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *,
                                  int *);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *, int);
```

To design our specialization of the mutex class, we need to add the ability to perform read locks and write locks. The pthreads library has a read/write mutex variable and attribute variable:

pthread\_rwlock\_t and pthread\_rwlockattr\_t

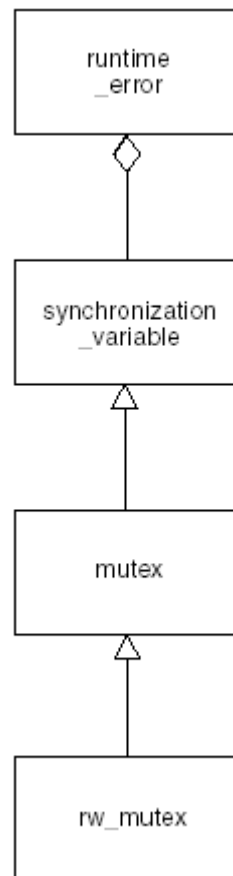
These variables are used in conjunction with the 11 pthread\_rwlock() functions. We use our interface class rw\_mutex to encapsulate the pthread\_rwlock\_t and pthread\_rwlockattr\_t variables and to wrap the Pthread read/write mutex functions.

**Example 11.9 Declaration of rw\_mutex class that contains pthread\_rwlock\_t and pthread\_rwlockattr\_t objects.**

```
class rw_mutex : public mutex{
protected:
    struct pthread_rwlock_t *RwLock;
    struct pthread_rwlockattr_t *RwLockAttr;
public:
    //...
    int read_lock(void);
    int write_lock(void);
    int try_readlock(void);
    int try_writelock(void);
    //...
};
```

The rw\_mutex class inherits the mutex class. [Figure 11-3](#) shows the class relationships between our rw\_mutex class, mutex class, synchronization\_variable class and our runtime\_error class.

**Figure 11-3. The class relationships between rw\_mutex, mutex, synchronization\_variable, and runtime\_error classes.**



So far we have a somewhat narrow interface. We are only interested in providing the core minimum set of attributes and characteristics needed to generalize our mutex class using the mutex types and functions from the Pthread library. However, once we are done with this narrow interface for this mutex class we use that interface as a basis for our semi-fat interface. The narrow interface typically is used with classes that are all related through inheritance in some way. The fat interfaces tend to be used with classes that are related by functionality and not by inheritance. We can use the interface class to simplify the interface on classes or functions that belong to different libraries but have similar functionality. The interface class will provide the programmer with a consistent look and feel. We take each of the libraries or classes that have similar functionality, collect all of the common functions and variables, then generalize that functionality into a large class that contains all of the required functions and attributes. This will define a class with a fat interface. However, if we just include the functions and data we are interested in, we (e.g., rw\_mutex class) then have a semi-fat interface. It has some of the advantages of the fat interface by allowing us to access objects that are only related by functionality and it restricts the set of methods the programmer has to deal with to the methods contained in the narrow interface class. This can be very important when integrating large function libraries like MPI and PVM with the POSIX facilities for concurrency. The combination of the MPI, PVM, and POSIX facilities represents hundreds of functions that all have very similar goals. Taking the time to streamline this functionality into interface classes will allow the programmer to reduce some of the complexity involved with parallel and distributed programming. Also, these interface classes become reusable components that support concurrency.

To see how we approach our semi-fat interface, let's provide an interface class for the POSIX semaphore. Although the semaphore is not part of the Pthread library, it certainly has similar uses within a multithreaded environment. However, it can be used in an environment that includes concurrently executing processes as well as threads. So in some cases it is a more general synchronization variable than our mutex class.

We might define our semaphore class in [Example 11.10](#) as:

**Example 11.10 Declaration of semaphore class.**

```
class semaphore : public synchronization_variable{
protected:
    sem_t *Semaphore;
public:
    //...
    int lock(void);
    int unlock(void);
    int trylock(void);
    //...
};
```

## Synopsis

<semaphore.h>

```
int sem_init(sem_t *, int, unsigned int);
int sem_destroy(sem_t *);

sem_t *sem_open(const char *, int, ...);
int sem_close(sem_t *);
int sem_unlink(const char *);
int sem_wait(sem_t *);
int sem_trywait(sem_t *);
```

```
int sem_post(sem_t *);
int sem_getvalue(sem_t *, int *);
```

Notice that it has the same interface as our mutex class. What's the difference? First there are several important POSIX semaphore functions. Although the interfaces of mutex and semaphore are the same, the implementation of the lock(), unlock(), trylock(), and so on functions will be calls to the POSIX semaphore functions. For instance:

**Example 11.11** Definitions of lock(), unlock(), and trylock() methods for the semaphore class.

```
int semaphore::lock(void)
{
    //...
    return(sem_wait(Semaphore));
}
int semaphore::unlock(void)
{
    //...
    return(sem_post(Semaphore));
}
```

So the lock(), unlock(), trylock(), and so on, functions will wrap POSIX semaphore functions instead of Pthread functions. It is very important to note that a semaphore and a mutex are not the same thing. However, they can be used in similar situations. Often from the point of view of instructions that are implementing parallelism, the lock() and the unlock() mechanisms serve the same purpose. [Table 11-2](#) shows some of the fundamental differences between a mutex and a semaphore.

While the differences in semantics in [Table 11-2](#) are important, they are often not enough to justify a completely different interface to semaphore and mutexes. Therefore, we keep our lock(), unlock(), and trylock() semi-fat interface with the caveat that the programmer must know the differences between a mutex and a semaphore. This is similar to the situation that arises with the fat interfaces of the container classes such as deque, queue, set, multiset, and so on. The container classes are related by interface but their semantics are different in certain areas. Using the notion of an interface class, synchronization components can be designed for mutexes, condition variables, read/write mutexes, and semaphores. Once we have these components, we can design concurrency-safe container classes, domain classes, and framework classes. We can also use the interface classes to provide a single interface to different versions of the same function library, where both versions need to be used within the same application for some reason. Sometimes the interface class can be used to bridge the gap between deprecated, obsolete functions and new functionality. We often want to insulate the application programmer from the difference between operating systems. When the System V semaphores or POSIX semaphores are used, the programmer can be provided with a consistent API using an interface class.

**Table 11-2. Fundamental Differences between Mutexes and Semaphores**

### **Characteristics of Mutexes**

### **Characteristics of Semaphores**

Mutexes and condition variables are shared between threads.

Semaphores are typically shared between processes, but may also be shared between threads.

A mutex is unlocked by the same threads that locked it.

A semaphore post can be performed by other than the original thread or process that held it.

A mutex is either locked or unlocked.

A semaphore is managed by its reference count state.

The POSIX standard includes named semaphores.

## **11.3 Maintaining the Stream Metaphor**

Besides using interface classes to simplify and to create new fat interfaces for the parallel and message-passing libraries, we may also want to extend existing interfaces. For instance, the object-oriented stream metaphor can be extended to pipes, fifos, and the message passing libraries like PVM and MPI. These components are used to accomplish IPC (Inter-Process Communication), ITC (Inter-Thread Communication), and in some cases OTOC (Object-to-Object Communication). When communication occurs between concurrently executing threads or processes, then the communication channel may be a critical section. That is, if two or more routines are attempting to update the same pipe, fifo, or message buffer at the same time, then a race condition is present. If we are going to extend the object-oriented stream interface to include components from the PVM or MPI library, then we need to make sure that the streams we design are concurrency safe. Here is where our synchronization components that were designed as interface classes come in handy. Let's look at a simple `pvm_stream` class.

**Example 11.12 Declaration of `pvm_stream` class that inherits `mios` class.**

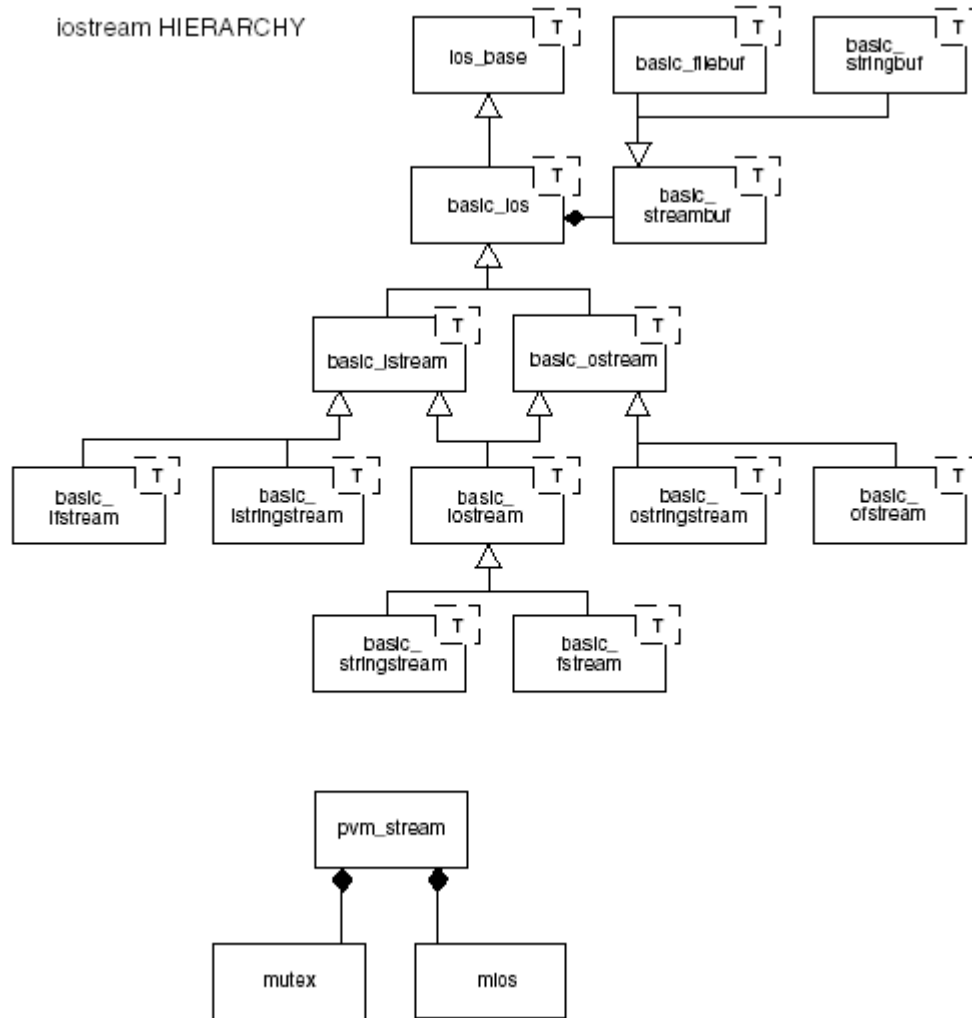
```
class pvm_stream : public mios{
protected:
    int TaskId;
    int MessageId;
    mutex Mutex;
    //...
public:
    void taskId(int Tid);
    void messageId(int Mid);
    pvm_stream(int Coding = PvmDataDefault);
    void reset(int Coding = PvmDataDefault);
    pvm_stream &operator<<(string &Data);
    pvm_stream &operator>>(string &Data);
    pvm_stream &operator>>(int &Data);
    pvm_stream &operator<<(int &Data);
    //...
};
```

This stream class will be used to encapsulate the state of the active buffer in a PVM task. The inserter operator << and the extractor operator >> will be used to send and retrieve messages between PVM processes. Here we only show operators for string and int types. The interface for this class is far from complete. Since this class could possibly be used with any datatype, we have to expand the inserter and extractor definitions. Since we plan to use the pvm\_stream class within a multithreaded program, we want to make sure that the pvm\_stream object is thread safe. Therefore, we include a mutex class as a member of our pvm\_stream class. The pvm\_stream class also encapsulates the active buffer for the PVM task. The stream can direct the message to a particular PVM task. The goal is to use the ostream and istream classes as a guide for the type of functionality that the pvm\_stream class should have. Recall that istream and ostream classes are translator classes. They translate datatypes into generic streams of bytes for output and from generic streams of bytes to specific datatypes on input. Using the istream and ostream classes, the programmer does not have to get bogged down in details of what datatype is being inserted or extracted from a stream. We want the pvm\_stream to behave in the same manner. The PVM has a different function for every type that needs to be packed into or unpacked from a send or receive buffer. For instance:

```
pvm_pkdouble()  
pvm_pkint()  
pvm_pkfloat()
```

are used to pack doubles, ints, and floats. There are similar functions for the other datatypes that C++ uses. We would like to retain our stream metaphor where input and output is seen as a generic stream of bytes moving into or out of the program. Therefore, we define the << inserter operator and the >> extractor operator for every type we wish to exchange between PVM tasks. Furthermore, we also model the stream state after istream and ostream classes. The istream and ostream classes have an ios component that is used to hold the state of the stream. The stream may be in an error state, or the stream may be in one of several different numeric states such as octal, decimal, and hexadecimal. The stream may be in a good state, a locked state, an end-of-file state, and so on. The pvm\_stream class should have a component that maintains the state of the stream and should have methods that set, reset, and report the PVM stream state. Our pvm\_stream class contains a mios component for this purpose. The mios component maintains the state of the stream and the active send and receive buffer. [Figure 11-4](#) contains a class diagram showing the relationships between the major classes in the iostream class library and how the pvm\_stream class compares.

**Figure 11-4. The class diagram showing the relationship between the major classes in the iostream class library and the class diagram for the pvm\_stream class.**



Notice that the istream and ostream classes inherit an ios class. The ios class maintains the stream state and buffer state of the istream and ostream class. Our mios class plays the same role for the pvm\_stream class. The istream and ostream classes contain the definitions for the inserter << and extractor >> operators. The operators are defined by our pvm\_stream class. So although our pvm\_stream class is not related to the iostream classes by inheritance, it is related by interface. We use the interface of the iostream classes to dictate a semi-fat interface for the pvm\_stream and mios classes. Notice in [Figure 11-4](#) that the mios class is inherited by the pvm\_stream class. We want to maintain the stream metaphor with the pvm\_stream class. The notion of an interface class is used to accomplish this.

### 11.3.1 Overloading the <<, >> Operators for PVM Streams

Let's take a look at how the << inserter operators and >> extractor operators are defined for the pvm\_stream class. The << inserter operator is used to wrap the pvm\_send and pvm\_pk routines. A method definition that looks something like:



**Example 11.13 Definition of <<operator for the pvm\_stream class.**

```
pvm_stream &pvm_stream::operator<<(int Data)
{
    //...
    reset();
    pvm_pkint(&Data,1,1);
    pvm_send(TaskId,MessageId);
    //...
    return(*this);
}
```

is provided for each type that will be used with the pvm\_stream class. The reset() method is inherited from the mios class. This method is used to clear or initialize the send buffer. TaskId and MessageId are data members of the pvm\_stream class and are set with the taskId() and messageId() methods. The inserter method allows us to send data to a PVM task with the standard stream notation:

```
int Value = 2004;
pvm_stream MyStream;
//...
MyStream << Value;
//...
```

The >> extractor operators are used in a similar manner to receive messages from PVM tasks. The >> operator is used to wrap the pvm\_rcv() and pvmupk routines. Extractor operators can be defined as:

**Example 11.14 Definition of operator >> for the pvm\_stream class.**

```
pvm_stream &pvm_stream::operator>>(int &Data)
{
    int BufId;
    //...
    BufId = pvm_rcv(TaskId,MessageId);
    StreamState = pvm_upkint(&Data,1,1);
    //...
    return(*this);
}
```

This type of definition will allow us to receive messages from PVM tasks using the extractor operator.

```
int Value;
pvm_stream MyStream;

MyStream >> Value;
```

Because the operator returns a reference to the pvm\_stream, the insertion and extraction operators may be strung together:

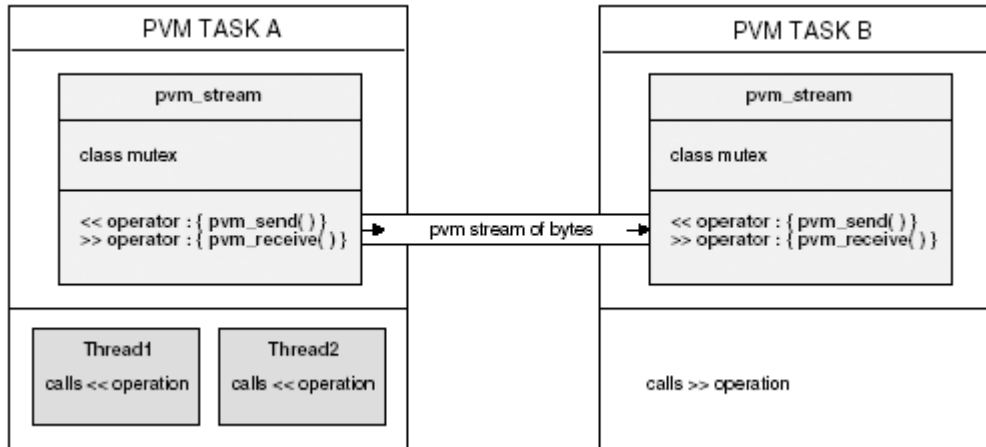
```
Mystream << Value1 << Value2;
Mystream >> Value3 >> Value4;
```

Using this syntax, the programmer is isolated from the more cumbersome syntax of the pvm\_send, pvm\_pk, pvm\_upk, and pvm\_rcv routines and the more familiar object-oriented stream metaphor can be used. In this case, the stream represents a message buffer and the items that are inserted and extracted from the streams represent messages that are being exchanged between PVM processes. Recall that each PVM process has a separate address space. So not only do the << insertion and >> extraction operators disguise the pvm\_send and pvm\_rcv calls, they also mask the underlying socket

activity. Since the `pvm_stream` class might be used in a multithreaded environment, the insertion and extraction operators need to be thread safe.

The class diagram in [Figure 11-4](#) shows that the `pvm_stream` class contains a `mutex` class. The `mutex` class can be used to protect the critical sections that are present in the `pvm_stream` class. The `pvm_stream` class encapsulates access to the send buffer and the receive buffer. [Figure 11-5](#) shows how threads and the `pvm_stream` class interact with the `pvm_send` and `pvm_receive` buffers.

**Figure 11-5.** The interaction between the threads, the `pvm_stream` class, and the `pvm_send` and `pvm_receive` buffers.



Not only are the send and receive buffers critical sections, the `mios` class used to store the state of the `pvm_stream` class is also a critical section. The `mutex` class can be used to protect this component as well.

The `Mutex` object can be used in the call to the insertion and extraction operators.

**Example 11.15 Definition of operator << and operator >> for the `pvm_stream` class.**

```
pvm_stream &pvm_stream::operator<<(int Data)
{
    //...
    Mutex.lock();
    reset();
    pvm_pkint(&Data,1,1);
    pvm_send(TaskId,MessageId);
    Mutex.unlock();
    //...
    return(*this);
}

pvm_stream &pvm_stream::operator>>(int &Data)
{
    int BufId;
    //...
    Mutex.lock();
    BufId = pvm_rcv(TaskId,MessageId);
    StreamState = pvm_upkint(&Data,1,1);
    Mutex.unlock();
    //...
    return(*this);
}
```

This kind of scheme can be used to make the `pvm_stream` class thread safe. We don't show the exception handling code or the extra processing that would be included to prevent indefinite postponement or deadlock. The idea here is to focus on the components and architectures that can be used to support concurrency. The mutex interface class and the `pvm_stream` class can be reused and both support concurrency programming. For our purposes, the `pvm_stream` objects are assumed to be used by the receiving and the sending PVM task. However, this is not a strict requirement. In order for the user to use the `pvm_stream` class concept with user-defined classes, the insertion operator (`<<`) and the extraction operator (`>>`) must be defined for the user-defined class.

## 11.4 User-Defined Classes Designed to Work with PVM Streams

To see how the user-defined class can be used with the `pvm_stream` class, we will improve on the PVM palette producing routines introduced in [Chapter 6](#). The palette class represents a simple collection of colors. For convenience, we store the colors in a `vector<string>` named `Colors`.

We begin by declaring a `spectral_palette` class that contains friend declarations for the `<<` inserter and `>>` extractor operators.

### Example 11.16 Declaration of `spectral_palette` class.

```
class spectral_palette : public pvm_object{
protected:
    //...
    vector<string> Colors;
public:
    spectral_palette(void);
    //...
    friend pvm_stream &operator>>(pvm_stream &In,
                                   spectral_palette &Obj);
    friend pvm_stream &operator<<(pvm_stream &Out,
                                   spectral_palette &Obj);
    //...
};
```

Notice this `spectral_palette` class in [Example 11.16](#) inherits a `pvm_object` class. The `pvm_object` class provides the `spectral_palette` class with member access to a task id and message id. Recall that the task id and message id are used in many PVM routines. With the definition of the `<<` operator and `>>` operator, `spectral_palette` objects can be sent between concurrently executing PVM tasks. The technique used with the `spectral_palette` class is a simplified approach that can be used with any user-defined class. Since the `pvm_stream` class will have operators for the built-in datatypes and for containers that hold built-in datatypes, a user-defined class need only define the `<<` operator and `>>` operator to translate its representation to either a built-in datatype or a standard container that holds built-in datatypes. For instance, the `<<` operator for the `spectral_palette` class is defined in [Example 11.17](#).

**Example 11.17 Definition of operator<< for the spectral\_palette class.**

```
pvm_stream &operator<<(pvm_stream &Out, spectral_palette &Obj)
{
    int N;
    string Source;
    for(N = 0;N < Obj.Colors.size();N++)
    {
        Source.append(Obj.Colors[N]);
        if(N <Obj.Colors.size() - 1){
            Source.append(" ");
        }
    }
    Out.reset();
    Out.taskId(Obj.TaskId);
    Out.messageId(Obj.MessageId);
    Out << Source;
    return(Out);
}
```

Let's closely examine the definition of this insertion operation in [Example 11.17](#). Since the `pvm_stream` class only works with built-in types, the goal of the user-defined `<<` operator is to translate the user-defined object into a sequence of user-defined datatypes. This translation process is one of the primary responsibilities of the stream classes. Here, the `spectral_palette` class will be translated into a string of colors separated by a blank. The list of colors is stored in a string named `Source`. This translation process allows the class to be used with the `pvm_stream <<` operator that has been defined for the string datatype. Once these operators are defined, the programmer's API is considerably more consistent than it would be if the native versions of the Pthread library, POSIX semaphore library, and MPI library routines were called. A `spectral_palette` object can be sent to another PVM task using an `<<` insertion operator as:

**Example 11.18 Using the pvm\_stream and spectral\_palette objects.**

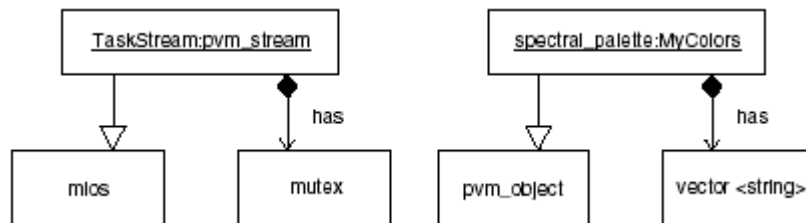
```
pvm_stream TaskStream;
spectral_palette MyColors;

//...
TaskStream.taskId(20001);
TaskStream.messageId(1);
//...
TaskStream << MyColors;
//...
```

The `MyColors` object is sent to the appropriate PVM task. [Figure 11-6](#) contains the components used to support the `TaskStream` and `MyColors` objects. Each component in [Figure 11-6](#) can be refined and optimized individually. Each layer provides an additional level of insulation from the complexity of these components. At the highest level the programmer is only concerned about the application domain. This high level of abstraction allows the programmer to naturally represent the parallelism that the application domain requires without getting bogged down in syntax and complicated function call sequences. The components in [Figure 11-6](#) represent the beginnings of a class library that can be used for PVM programs and multithreaded PVM programs. These same techniques can be used to communicate between concurrently executing tasks that are not part of a PVM. There are many applications that require concurrency but that do not need the complete machinery of the PVM environment. For these applications the `exec()`, `fork()`, or `pvm_spawn()` functions are sufficient. Applications that only require a few concurrently executing processes and client-server applications are

good examples. Interprocess communication is also required for these non-PVM or non-MPI applications. We would also like to maintain the stream metaphor for concurrently executing processes created with the fork-exec sequence or `pvm_spawn`. The notion of the object-oriented stream can be extended to cover pipes and fifos.

**Figure 11-6. The components used to support the TaskStream and MyColors objects.**



## 11.5 Object-Oriented Pipes and fifos as Low-Level Building Blocks

To arrive at a design for object-oriented pipes, we start with basic characteristics and behavior that all pipes have in common. A pipe is a channel of communication between two or more processes. In order for the processes to communicate, they will transmit some sort of information between them. The information may represent data or commands to be performed. Typically, the information is translated into a sequence of data and inserted into the pipe and retrieved by a process on the other side of the pipe. The data is reassembled into meaningful data by the retrieving process. Whatever the data represents, there must be somewhere to store the data while it is in transit from one process to another. We call the storage area for the information a data buffer. Insertion and extraction operations are needed to place data into and extract data from the buffer. Before any insertion into the data buffer or extraction from a data buffer can begin, the data buffer must first exist. Object-oriented piping facilities must support an operation that creates and initializes a data buffer. Once the communications between processes have been completed, the data buffer used to hold the information will no longer be necessary. This means that our object-oriented pipe must be able to remove the data buffer after it is no longer needed. This suggests there are at least five basic components that any object-oriented piping facility should have:

- Buffer
- Insertion operation
- Extraction operation
- Creation/initialization operation
- Destruction operation

In addition to these five basic components, a pipe will have two ends. One end of the pipe will be for inserting data. The other end of the pipe will be for extracting data. These two ends can be accessed from different processes. To complete our notion of a pipe we must include an input port and an output port, which could be connected to separate processes. This gives seven basic components needed to describe our object-oriented pipe:

- Input port
- Output port
- Buffer

- Insertion operation
- Extraction operation
- Creation/initialization operation
- Buffer destruction operation

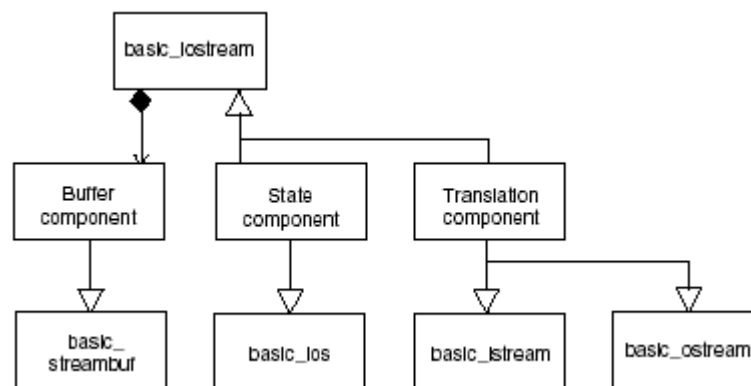
These components represent minimal core requirements for our description of a pipe. Once we have the basic components, we can identify how existing system APIs or data structures can be used to help us design an object-oriented pipe. In the same way that we use encapsulation and operator overloading to design a `pvm_stream`, we use the same techniques to wrap the pipe and fifo functions.

Notice that five of the seven basic components are common to many basic I/O data structures and container types. Most UNIX/Linux file services support:

- Buffers
- Buffer insertion operations
- Buffer extraction operations
- Buffer creation operations
- Buffer destruction operations

We use the notion of C++ interface classes to encapsulate the functionality provided by UNIX/Linux system services. We build object-oriented versions of the input/output services. Whereas we had to start from scratch with the `pvm_stream` class for the PVM library, here we can take advantage of the existing C++ standard library and the `iostreams` class library supports an object-oriented model of input and output streams. Recall that the `iostreams` class library supports an object-oriented model of input and output streams. Furthermore, this object-oriented library has support for the data buffer notion and all the operations upon the data buffer. [Figure 11-7](#) shows a simple class diagram of the `iostream` class.

**Figure 11-7. Class diagram for the major components of the `iostream` class.**



The major components of the `iostream` class can be described by three kinds of classes: a buffer component, a translation component, and a state component (see Hughes & Hughes, 1999). The buffer component is used as a holding area for bytes that are in transit. The translation component is responsible for translating anonymous sequences of bytes into the appropriate datatypes and data structures, and for translating data structures and data-types in anonymous sequences of bytes. The translation component is responsible for providing the programmer with a stream of bytes metaphor where all I/O regardless of source or destination is treated as a stream of bytes. The state component encapsulates the state of the object-oriented stream. The state component maintains what type of formatting is applicable to the data bytes that are in the buffer component. The state component also

maintains whether a stream has been opened in the append mode, create mode, exclusive read, exclusive write, or whether numbers will be interpreted as hexadecimal, octal, or binary. The state component also can be used to determine the error state of I/O operations on the buffer component. By querying the state component the programmer can determine if the buffer component is in a good or bad state. These three components are objects and can be used together to form a complete object-oriented stream, or separately as support objects in other tasks.

Five of the basic requirements for our pipe are already implemented in the `iostreams` class library. All we need to add is the notion of the input port and output port. To do this, we can examine the system services that support the use of pipes. The UNIX/Linux system calls create a pipe.

**Example 11.19 Using system call to create a pipe.**

```
int main(int argc, char *argv[])
{
    //...
    int Fd[2];
    pipe(Fd);
    //...
}
```

The `pipe` function call is used to create a pipe data structure, which can be used between parent and child processes to communicate. If the call to `pipe` is successful, it will return two file descriptors. File descriptors are integers used to identify successfully opened files. In this case, the descriptors are stored in the array `Fd`. `Fd[0]` will be open for reading, and `Fd[1]` will be open for writing. Once these two file descriptors are created, they can be used with regular `read( )` and `write( )` functions. The `write( )` function will cause data to be inserted into the pipe via `Fd[1]`, and the `read( )` function will cause data to be extracted from the pipe via `Fd[0]`. Because the `pipe()` function returns file descriptors, access to a pipe can be accomplished using system file services. The `sysconf(_SC_OPEN_MAX)` system call can be used to determine the maximum allowable file descriptors open by a process. The `pathconf(_PC_PIPE_BUF)` call can be used to find the size of the pipe.

These two file descriptors represent our logical input port and output port, respectively. We also use these two file descriptors to provide links to the `iostream` class library. Specifically, they provide a link to the buffer class. The buffer component of the `iostream` classes has three families of classes. [Table 11-3](#) lists the three types of buffer classes and their descriptions.

**Table 11-3. Three Types of Buffer Classes**

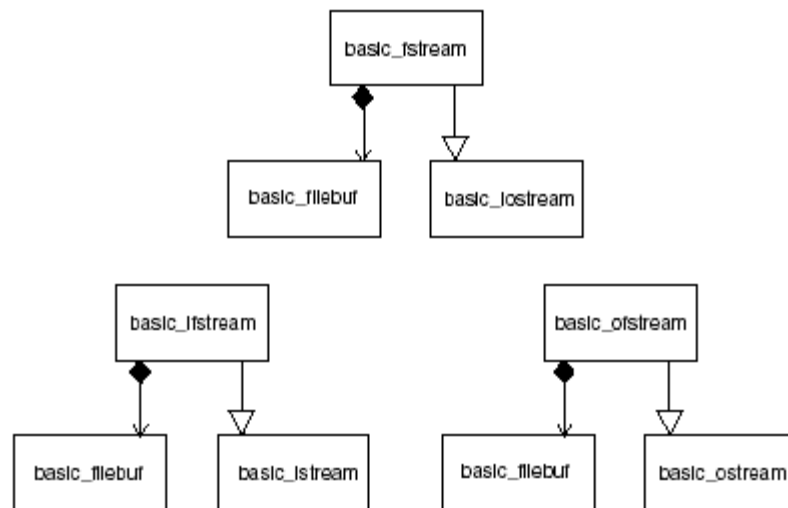
**Types of Buffer Description  
Classes**

<code>basic_streambuf</code>	Describes the behavior of various stream buffers in order to control input and output sequences of characters.
<code>basic_stringbuf</code>	Associates input and output sequences with a sequence of arbitrary characters that can be initialized from or made available as a string object.
<code>basic_filebuf</code>	Associates input and output sequences of characters with a file.

We are interested in the `filebuf` class. Whereas the `basic_streambuf` class is used as an object-oriented

buffer in I/O from standard in and standard out, and the `basic_stringbuf` class is used for object-oriented memory buffers, the `filebuf` class is used as object-oriented buffers for files. By examining the interface for the `filebuf` class and the interface for its translator classes, `ifstream`, `ofstream`, and `fstream`, we can find a way to connect the file descriptors returned from the `pipe()` system call to the `iostream` objects. [Figure 11-8](#) shows the class diagrams for the `fstream` family of classes.

**Figure 11-8. The class diagrams for the `fstream` family of classes.**



Notice that the `ifstream`, `ofstream`, and `fstream` classes all contain the `filebuf` class. Therefore, we can use any class from the `fstream` family of classes to help us in creating object-oriented pipe facilities. We can connect the file descriptors returned by the `pipe()` system call either through constructors, or through the `attach()` member function.

## Synopsis

```

#include <fstream>

// UNIX systems
ifstream(int fd)
fstream(int fd)
ofstream(int fd)

// gnu C++
void attach(int fd);

```

### 11.5.1 Connecting Pipes to `iostream` Objects Using File Descriptors

There are three `iostream` classes that we can use to connect to a pipe. They are `ifstream`, `ofstream`, and `fstream`. An `ifstream` object is used for input and an `ofstream` object is used for output. An `fstream` object can be used for both input and output. Although direct support for file descriptors and the `iostreams` is not yet part of the ISO standard, most UNIX and Linux C++ environments support `iostream` access to file descriptors. The GNU C++ `iostreams` library supports a file descriptor in one of the `ifstream`, `ofstream`, and `fstream` constructors and with the `attach()` method of the `ifstream` and `ofstream` classes. UNIX compilers such as Sun's C++ compiler supports file descriptors through one of the `ifstream`, `ofstream`, and `fstream` constructors. So the sequence of code:

```

//...

```



```

int Fd[2];
Pipe(Fd);
ifstream IPipe(Fd[0]);
ofstream OPipe(Fd[1]);

```

will create two object-oriented pipes. IPipe will be an input stream and OPipe will be an output stream. Once these streams are created they can be used to communicate between concurrently executing processes using the stream metaphor and the inserter << and >> extractor operators. For the C++ environments that support the attach() method, the file descriptor can be attached to an ifstream, ofstream, or fstream object using the syntax:

**Example 11.20 Creating a pipe and using the attach() function.**

```

int Fd[2];
ofstream OPipe;
//...
pipe(Fd);
//...
OPipe.attach(Fd[1]);
//...
OPipe << Value << endl;

```

This usage of object-oriented pipes assumes the existence of some child process that can read the pipe. [Program 11.1](#) uses the fork command to create two processes. The parent process sends a value to the child process using an iostreams-based pipe.

**Program 11.1**

```

1 #include <unistd.h>
2 #include <iostream.h>
3 #include <fstream.h>
4 #include <math.h>
5 #include <sys/wait.h>
6
7
8
9
10 int main(int argc, char *argv[])
11 {
12
13 int Fd[2];
14 int Pid;
15 float Value;
16 int Status;
17 if(pipe(Fd) != 0){
18     cerr << "Error Creating Pipe " << endl;
19     exit(1);
20 }
21 Pid = fork();
22 if(Pid == 0){
23     ifstream IPipe(Fd[0]);
24     IPipe >> Value;
25     cout << "Value Received From Parent " << Value << endl;
26     IPipe.close();
27 }
28 else{
29     ofstream OPipe(Fd[1]);
30     OPipe << M_PI << endl;

```

```
31     wait(&Status);
32     OPipe.close();
33
34 }
35
36 }
```

Recall that when a `fork()` call is made, the return value of 0 belongs to the child process. In [Program 11.1](#) the pipe is created on line 17. On line 29 the parent process opens the pipe for writing. The file descriptor `Fd[1]` is the write end of the pipe. This end of the pipe is attached to an `ofstream` object through the constructor on line 29. The read end of the pipe is attached to an `ifstream` object on line 23. The child process opens the pipe for reading. The child process has access to the file descriptor because along with the parent's environment, file descriptors are also inherited. Therefore, any files that are open in the parent will be opened in the child unless explicit instructions are given to the operating system using the `fcntl` system call. Besides open files being inherited, the position markers within the files remain where they are during the spawning of the child process so that the child process also has access to the position marker. When the position is moved in the parent, the marker in the child process is also moved. In this case, we were able to accomplish the stream metaphor without creating an interface class. Simply by attaching the pipe's file descriptors to the `ofstream` and `ifstream` objects we are able to use the `<< inserter` and `>> extractor` operators. Likewise, any class that has the `>> extraction` or `<< insertion` operators defined can be extracted from or inserted into the pipe without requiring any further work from the programmer. The parent inserts the value `M_PI` into the pipe on line 30. The child extracts the value from the pipe using the `>>` operator on line 24. The details for executing and compiling this program are contained in [Program Profile 11.1](#).

## Program Profile 11.1

Program Name

```
program11-1.cc
```

Description

[Program 11.1](#) demonstrates the use of an object-oriented stream metaphor with anonymous system pipes. The program uses the `fork()` program to create two processes that will communicate using the `<< inserter` and `>> extractor` operators.

Headers Required

```
<wait.h>, <unistd.h>, <iostream.h>, <fstream.h>, <math.h>
```

Compile and Link Instructions

```
c++ -o program11-1 program11-1.cc
```

Test Environment

Solaris 8, SuSE Linux 7.1

Execution Instructions

```
./program11-1
```

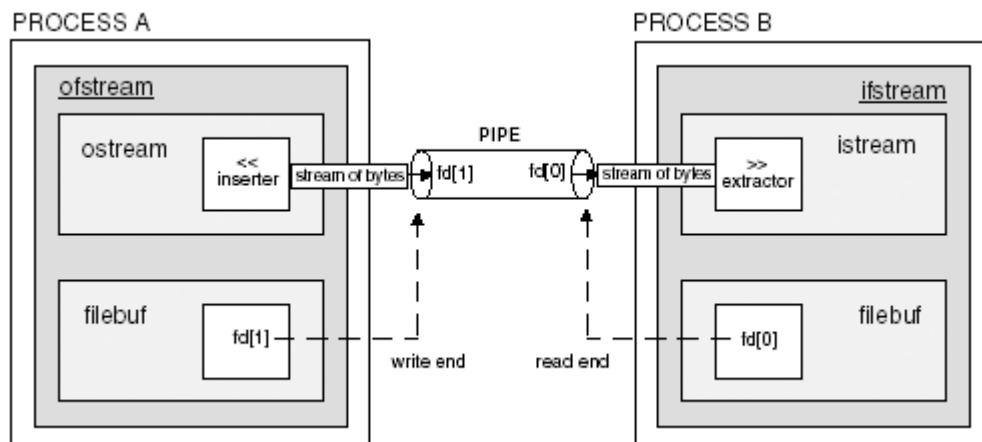
The gnu C++ compiler also supports the `attach()` method. We could use this method to connect the file descriptors to the `ifstream` and `ofstream` objects. For instance:

**Example 11.21 Connecting file descriptors to an `ofstream` object.**

```
int main (int argc, char *argv[])
{
    int Fd[2];
    ofstream Out;
    pipe(Fd);
    Out.attach(Fd[1]);
    //...
    // Interprocess Communication
    //...
    Out.close( );
}
```

The `Out.attach(Fd[1])` call attaches an `ofstream` object to a pipe file descriptor. Now any information that is inserted into the `Out` object is actually being written to a pipe. Using extractors and insertors to perform automatic format translation is a major advantage of using the `fstream` family of classes in conjunction with the pipe communication. The ability to use userdefined extractors and insertors removes some of the difficulty encountered with pipe programming. So instead of requiring the explicit enumeration of data sizes of everything written and read from the pipe we use the number of elements to control read/write access, which makes the entire process simpler. In addition, this cost savings makes the parallel programming efforts easier. The technique we recommend is to use architecture to support a divide-and-conquer approach to parallelization. Once the correct components are in place, the programming becomes easier. For instance, since the pipe is tied to `ofstream` and `ifstream` objects, we are able to use the information retained by the `ios` component to determine the state of the pipe. The translation components of the `istream`s can be used to perform automatic conversions as the data is inserted into one end of the pipe and extracted out of the other end. Using the pipes with the `istream`s also allows the programmer to integrate the standard containers and algorithms with pipe inter-process communication. [Figure 11-9](#) shows the relationship between the `ifstream`, `ofstream`, extractor, insertor, pipe, and the insertor and extractor when `istream`s are used for interprocess communication.

**Figure 11-9. The relationships between `ifstream` and `ofstream` objects, pipe, and the insertor and extractor when the `istream`s are used for interprocess communication.**



The `fstream` family of classes can also use the `read()` and `write()` member functions to read data to a

pipe, and write data from a pipe.

### 11.5.2 Accessing Anonymous Pipes Using the ostream\_iterator

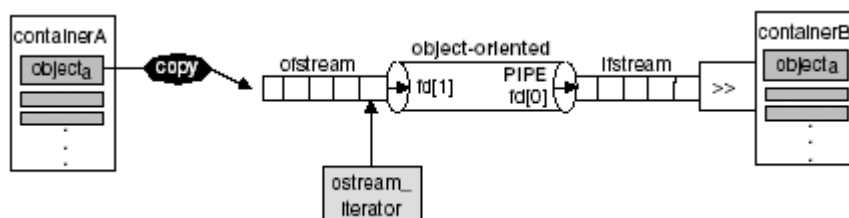
We can also use the pipe with the ostream\_iterator and istream\_iterator. These iterators are generic object-oriented pointers. The ostream\_iterator will allow you to transfer entire containers (i.e., lists, vectors, sets, queues, etc.) across the pipe. Without the use of iostreams and ostream\_iterator, transferring containers of objects would be a very tedious and error-prone process. [Table 11-4](#) lists the set of operations that are available on the ostream\_iterator and istream\_iterator class.

**Table 11-4. Set of Operations Available on the ostream\_iterator and istream\_iterator**

Iterators	Operations	Description
istream_iterator	a == b	Equivalence relation
	a != b	Nonequivalence relation
	*a	Dereference
	++ r	Preincrementation
	r ++	Postincrementation
ostream_iterator	++ r	Preincrementation
	r ++	Postincrementation

Typically, these iterators are used with the iostreams classes and the standard algorithms. The ostream\_iterator is a sequential write-only iterator. Once an item has been accessed, the programmer cannot go back to it without starting the iteration over. The pipe is treated like a sequence container when used with these iterators. This means that when the pipe is connected to the iostreams through ostream\_iterator and the file descriptors, we can apply standard algorithm type processing to the input from a pipe or the input to a pipe. The reason these iterators can be used in conjunction with pipes is the connection between the iterators and the iostream classes. The diagram in [Figure 11-10](#) shows the relationship between the I/O iterators and the iostream classes.

**Figure 11-10. The relationship between the I/O iterators and the iostream classes.**



[Figure 11-10](#) also shows how these classes interact with the notion of our object-oriented pipe. Let's

take a close look at how the `ostream_iterator` is used with an `ostream` object. If a pointer is incremented, we expect it to point at the next location in memory. When the `ostream_iterator` is incremented, it moves or points to the next position in the output stream. When we assign a value to a dereferenced pointer, we are placing that value at the location that the pointer points to. When we assign a value to an `ostream_iterator`, we are placing that value in the output stream. If that output stream is connected to `cout`, then the value will be displayed on the standard out. If we declare an `ostream_iterator` object such as:

```
ostream_iterator<int> X(cout, "\n");
```

Then `X` is an object of type `ostream_iterator`. The increment operation:

```
X++;
```

causes `X` to move to the next position in the output stream. The statement:

```
*X = Y;
```

causes `Y` to be displayed on standard out. This is because the assignment operator `=` has been overloaded to use an `ostream` object. The declaration:

```
ostream_iterator<int> X(cout, "\n");
```

caused `X` to be constructed with a `cout` as the stream. The other argument in the constructor is the delimiter that will automatically be placed after every `int` that is inserted into the stream. The declaration for the `ostream_iterator` looks like this:

#### Example 11.22 Declaration of the `ostream_iterator` class.

```
template <class _Tp>
class ostream_iterator {
protected:
    ostream* _M_stream;
    const char* _M_string;

public:
    typedef output_iterator_tag iterator_category;
    typedef void value_type;
    typedef void difference_type;
    typedef void pointer;
    typedef void reference;

    ostream_iterator(ostream& __s) : _M_stream(&__s), _M_string(0) {}
    ostream_iterator(ostream& __s, const char* __c)
        : _M_stream(&__s), _M_string(__c) {}
    ostream_iterator<_Tp>& operator=(const _Tp& __value) {
        *_M_stream << __value;
        if (_M_string) *_M_stream << _M_string;
        return *this;
    }
    ostream_iterator<_Tp>& operator*() { return *this; }
    ostream_iterator<_Tp>& operator++() { return *this; }
    ostream_iterator<_Tp>& operator++(int) { return *this; }
};
```

The constructor for the `ostream_iterator` accepts a reference to an `ostream` object. The `ostream_iterator` class has an aggregate relationship with the `ostream` class. The `istream_iterator` has just the opposite use

of the `ostream_iterator`. It is used with `istream` objects instead of `ostream` objects. If `istream_iterator` and `ostream_iterator` objects are connected to `iostream` objects that in turn are connected to pipe file descriptors, then every time the `istream_iterator` is incremented, the pipe is being read, and every time the `ostream_iterator` is incremented, the pipe is being written. To demonstrate how these components work together, we have two programs: [Programs 11.2](#) and [11.2b](#) that use anonymous pipes to communicate. [Program 11.2](#) is the parent and [Program 11.2b](#) is the child. The parent uses the `fork()` and `execl()` system calls to create the child process. Although file descriptors are inherited by the child, their values are immediately available to [Program 11.2b](#) because an `execl()` call has been made.

### Program 11.2

```
10 int main(int argc, char *argv[])
11 {
12
13     int Size,Pid,Status,Fd1[2],Fd2[2];
14     pipe(Fd1); pipe(Fd2);
15     stringstream Buffer;
16     char Value[50];
17     float Data;
18     vector<float> X(5,2.1221), Y;
19     Buffer << Fd1[0] << ends;
20     Buffer >> Value;
21     setenv("Fdin",Value,1);
22     Buffer.clear();
23     Buffer << Fd2[1] << ends;
24     Buffer >> Value;
25     setenv("Fdout",Value,1);
26     Pid = fork();
27     if(Pid != 0){
28         ofstream OPipe;
29         OPipe.attach(Fd1[1]);
30         ostream_iterator<float> OPtr(OPipe,"\n");
31         OPipe << X.size() << endl;
32         copy(X.begin(),X.end(),OPtr);
33         OPipe << flush;
34         ifstream IPipe;
35         IPipe.attach(Fd2[0]);
36         IPipe >> Size;
37         for(int N = 0; N < Size;N++)
38             {
39                 IPipe >> Data;
40                 Y.push_back(Data);
41             }
42         wait(&Status);
43         ostream_iterator<float> OPtr2(cout,"\n");
44         copy(Y.begin(),Y.end(),OPtr2);
45         OPipe.close();
46         IPipe.close();
47     }
48     else{
49         execl("./program11-2b","program11-2b",NULL);
50     }
51
52     return(0);
53 }
```

In lines 21 and 25, the `setenv()` system call is used to pass the values of file descriptors to the child. This is possible because the child process inherits the environment of the parent process. We can set

environment variables within a program using the `setenv()` system calls. So, in this case, we set:

```
Fdin=filedesc;  
Fdout=filedesc;
```

The child process then uses the `getenv()` system call to retrieve the values of `Fdin` and `Fdout`. The value in `Fdin` will be the read end of the pipe for the child and the value of `Fdout` will be the write end. Using the `setenv()` and `getenv()` system calls provide a simple form of IPC between parent and child processes. The pipes are created on line 14. On line 29 the parent attaches to one end of the pipe for writing using the `attach()` method. Once the attach is performed, any data inserted into the `OStream` object will be written to the pipe. An `ostream_iterator` is connected to the `OStream` object on line 30 using:

```
ostream_iterator<float> OPtr(OStream, "\n");
```

This causes the iterator `OPtr` to refer to `OStream`. The `"\n"` will be inserted as a delimiter after every insertion. Using `OPtr` we may insert any number of float values to the pipe. We can attach more than one iterator with different types to pipe. However, this does require that on the receiving end data is extracted using the appropriate types. In [Program 11.2](#) the number of elements is first inserted into the pipe using:

```
OStream << X.size() << endl;
```

The actual elements are sent using one of the C++ standard algorithms.

```
copy(X.begin(), X.end(), OPtr);
```

The `copy()` algorithm copies the contents of its container into the container associated with the target iterator. Here the target iterator is `OPtr`. `OPtr` is connected to the `OStream` so `copy()` causes the entire contents of the container to be written to the pipe in one line of code. This demonstrates how the standard algorithms can be used to help with the communication in parallel programming or distributed programming environments. Here the copy is sending information from one process to another process in a different address space. These processes are executing concurrently and the `copy()` algorithm makes the communication between the processes considerably easier. We emphasize this approach because everything that can be done to make the logic for a parallel or distributed program simpler should be done. Interprocess communication is one of the issues that complicates parallel and distributed programming. The C++ algorithms, the `iostreams`, and the `ostream_iterator` help to reduce that complexity. The flush manipulator on line 33 ensures that the data is moved along the pipe.

## Program Profile 11.2

Program Name

program11-2.cc

Description

Program uses the `iostreams` and the `ostream_iterator` to send the contents of a vector container over an anonymous pipe.

Headers Required

[\[View full width\]](#)

```
<algorithm>, <fstream>, <vector>, <iterator>, <stdlib.h>, <string
.h>,<unistd.h>
```

### Compile and Link Instructions

```
c++ -o program11-2 program11-2.cc
```

### Test Environment

SuSE Linux 7.1, 6.2

### Execution Instructions

```
./program11-2
```

In [Program 11.2b](#) on line 36 the child will get the number of elements to be retrieved from the pipe first and then it uses the istream object IPipe to retrieve the objects from the pipe.

### Program 11.2b

```
11 class multiplier{
12     float X;
13 public:
14     multiplier(float Value) { X = Value;}
15     float &operator()(float Y) { X = (X * Y);return(X);}
16 };
17
18
19 int main(int argc,char *argv[])
20 {
21     char Value[50];
22     int Fd[2];
23     float Data;
24     vector<float> X;
25     int NumElements;
26     multiplier N(12.2);
27     strcpy(Value,getenv("Fdin"));
28     Fd[0] = atoi(Value);
29     strcpy(Value,getenv("Fdout"));
30     Fd[1] = atoi(Value);
31     ifstream IPipe;
32     ofstream OPipe;
33     IPipe.attach(Fd[0]);
34     OPipe.attach(Fd[1]);
35     ostream_iterator<float> OPtr(OPipe,"\n");
36     IPipe >> NumElements;
37     for(int N = 0;N < NumElements;N++)
38     {
39         IPipe >> Data;
40         X.push_back(Data);
41     }
42     OPipe << X.size() << endl;
43     transform(X.begin(),X.end(),OPtr,N);
44     OPipe << flush;
45     return(0);
46
47 }
```



The child process retrieves the items from the pipe, inserts them into a vector class, and then performs a mathematical transformation on each element of the vector as it is sending it back to the parent. The mathematical transformation occurs on line 43 using the standard C++ transform algorithm and a user-defined multiplier class. The transform algorithm applies an operation to each element in a container and then inserts the results into the target container. Here, the target container is Optr, which is connected to an OPipe object. The required headers for [Program 11.2b](#) are shown in [Program Profile 11.2b](#).

## Program Profile 11.2b

Program Name

program11-2b.cc

Description

Child process that is launched by [Program 11.2](#). This program uses an ifstream object to receive the contents of a container that are sent from [Program 11.2](#). The program uses an ostream\_iterator object and the standard transform algorithm to send information back through the pipe to the parent.

Headers Required

[\[View full width\]](#)

```
<iostream>, <algorithm>, <fstream>, <vector>, <iterator>, <stdlib  
➔.h>, <string.h>, <unistd.h>
```

Compile and Link Instructions

```
c++ -o program11-2b program11-2b.cc
```

Execution Instructions

This program is spawned by [Program 11.2](#).

Although the iostreams classes, ifstream\_iterator, and ostream\_iterator make pipe programming easy, they do not change the behavior of the system pipe construct. The blocking issues and the issues concerning the correct order to open and close the pipes discussed in [Chapter 5](#) still apply. The underlying mechanisms of the same object-oriented programming techniques reduce the complexity of parallel and distributed programming.

### 11.5.3 fifos (Named Pipes), iostreams, and the ostream\_iterator Classes

The techniques we used to implement object-oriented anonymous pipes had two setbacks. First, any processes involved in interprocess communication need access to the file descriptors returned from the pipe() system call. So there is the issue of getting these file descriptors to all of the processes involved. This was straightforward because the processes that were created in [Programs 11.1](#), [11.2a](#), and [11.2b](#) had parent-child relationships, which leads us to the second problem. The processes using unnamed pipes need to be related, although this requirement could be subverted with a descriptor-passing scheme. The fifo (First In-First Out) structure is the solution to the problem. Its most important advantage is it can be accessed by unrelated processes. The processes do need to be on the same

machine but otherwise don't require any relationship. The processes may be running programs implemented in different languages using different programming paradigms (e.g., generic and object-oriented). Crowd computations and other peer-to-peer configurations can take advantage of fifo (sometimes called the named pipe) because in the UNIX and Linux environment the fifo has a user-defined file-name in the system and is somewhat of a permanent structure (in contrast to anonymous pipes). The fifo is a one-way structure, that is, the user of a named pipe in the UNIX environment should open it for either reading or writing, but not both. Named pipes created in the UNIX environment remain in the file system until they are explicitly removed using `unlink( )` from within a program, or some form of deletion at the command prompt such as the `rm` command. Named pipes are given the equivalent of a file-name when they are created. Any process that knows the name of a pipe and has the necessary access permissions can open, read, and write the pipe.

To connect the anonymous pipes to the `ifstream` and `ofstream` objects, we used the nonstandard file descriptor connection. File descriptors and the `iostreams` are not yet tightly coupled by the ISO C++ standard. We are a lot safer using the fifo. The special file type fifo is accessed through a filename in the file system. The connection with the C++ `ifstream` and `ofstream` classes is supported. So in the same way that we simplified IPC using `iostream` classes with the anonymous pipe, we make fifo access easy. So the fifo that has the same basic functionality as the anonymous pipe allows us to extend the communication between unrelated classes. However, each program involved will still have to know the names of the fifos. It seems like the same restriction as we encountered with the file descriptors. However, the fifo is a definite improvement. First, only the system determines what the available file descriptors are when the anonymous pipe is opened. This is out of the programmer's control. Second, there is a limit to the number of file descriptors the system has. Third, since fifos are user-defined names, there is no limit to the names that may be used. The file descriptors must belong to previously and successfully opened files. fifo names are just names. The fifo name is user-specified; file descriptors are system-specified. Filenames are associated with `ifstream`, `fstream`, and `ofstream` objects using either the constructor or the `open()` method. [Program 11.3a](#) uses the constructor to associate the `ofstream` and `ifstream` objects with the fifo.

### Program 11.3

```
14 using namespace std;
15
16 const int FMode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
17
18 int main(int argc, char *argv[])
19 {
20
21     int Pid,Status,Size;
22     double Value;
25     mkfifo("/tmp/channel.1",FMode);
26     mkfifo("/tmp/channel.2",FMode);
28     vector<double> X(100,13.0);
29     vector<double> Y;
30     ofstream OPipe("/tmp/channel.1",ios::app);
31     ifstream IPipe("/tmp/channel.2");
32     OPipe << X.size() << endl;
33     ostream_iterator<double> Optr(OPipe,"\n");
34     copy(X.begin(),X.end(),Optr);
35     OPipe << flush;
36     IPipe >> Size;
37     for (int N = 0;N < Size; N++)
38     {
39         IPipe >> Value;
40         Y.push_back(Value);
```

```

41  }
42
43  IPipe.close();
44  OPipe.close();
45  unlink("/tmp/channel.1");
46  unlink("/tmp/channel.2");
47  cout << accumulate(Y.begin(),Y.end(),-13.0) << endl;
48
49  return(0);
50 }

```

There are two fifos in [Program 11.3a](#). Recall that fifos are one-way communication components. So if processes are to exchange data, at least two fifos are necessary. In [Program 11.3a](#) the fifos are called channel.1 and channel2. Notice on line 16 the permissions flags that will be set for the fifos. These are the most generic settings for UNIX/Linux setting. These permissions indicate that the owner of the fifo has read and write access and all others have read-only access to the fifo. On line 30 channel.1 is open for output only. We could have also accomplished this by:

```
OPipe.open("/tmp/channel.1", ios::app);
```

This says that the fifo will be opened in append mode. [Program 11.3a](#) uses the copy() algorithm to insert the objects into the OPipe ostream object and indirectly into the fifo. We could also use a ostream object here if we declare it as:

```
ostream OPipe("/tmp/channel.1", ios::out | ios::app);
```

This restricts the communication to output only in append mode. If we don't use the ios::app flag, the ostream object on line 30 will make a failed attempt at creating the fifo. Unfortunately, this will not work. Creation of fifos is the province of the mkfifo() routine. Lines 40 and 41 [Program 11.3a](#) deletes the fifos from the file system. At this point in the processing any processes that happen to still have the fifo open will continue to be able to access it. However, the name will be removed. So those processes will not be able to call open() or construct any new ostream or ifstream objects based on the filename that has been unlinked. On lines 32-34, ostream\_iterator and ostream objects are used to insert items into the fifo. Notice that [Program 11.3a](#) does not do any forking and does not have any child processes to communicate with. [Program 11.3](#) depends on some other program to read from channel.1 or at least to write to channel.2. If there is no program executing at the time to access the fifo, then [Program 11.3a](#) will block. The implementation specifics are contained in [Program Profile 11.3a](#).

## Program Profile 11.3a

Program Name

program11-3.cc

Description

Uses an ostream\_iterator object and an ostream object to send a container object through a fifo. Extracts information from a fifo using an ifstream object.

Headers Required

<unistd.h>, <iomanip>, <algorithm>, <fstream.h>, <vector>,  
<iterator>

```
<sstream.h>, <stdlib.h>, <sys/wait.h>, <sys/types.h>,  
<sys/stat.h>  
<fcntl.h>, <numeric>
```

### Compile and Link Instructions

```
c++ -o program11-3 program11-3.cc
```

### Test Environment

SuSE Linux 7.1, gcc 2.95.2, Solaris 8, Sun Workshop 6

### Execution Instructions

```
./program11-3 & program11-3b
```

### Notes

Start [Program 11.3a](#) first. [Program 11.3b](#) has a sleep statement to account for the lack of real synchronization.

[Program 11.3b](#) reads from channel.1 and writes to channel.2.

### Program 11.3b Reads from channel1 and write to channel2.

```
10 using namespace std;  
11  
12 class multiplier{  
13     double X;  
14 public:  
15     multiplier(double Value) { X = Value;}  
16     double &operator()(double Y) { X = (X * Y);return(X);}  
17 };  
18  
19  
20 int main(int argc,char *argv[])  
21 {  
22  
23     double Size;  
24     double Data;  
25     vector<double> X;  
26     multiplier R(1.5);  
27     sleep(15);  
28     fstream IPipe("/tmp/channel.1");  
29     ofstream OPipe("/tmp/channel.2",ios::app);  
30     if(IPipe.is_open()){  
31         IPipe >> Size;  
32     }  
33     else{  
34         exit(1);  
35     }  
36     cout << "Number of Elements " << Size << endl;  
37     for(int N = 0;N < Size;N++)  
38     {  
39         IPipe >> Data;  
40         X.push_back(Data);
```

```

41 }
42 OPipe << X.size() << endl;
43 ostream_iterator<double> Optr(OPipe, "\n");
44 transform(X.begin(), X.end(), Optr, R);
45 OPipe << flush;
46 OPipe.close();
47 IPipe.close();
48 return(0);
49
50 }

```

Notice that [Program 11.3a](#) opens channel.1 for output and [Program 11.3b](#) opens channel.1 for input. Keep in mind that fifos are one-way communication mechanisms. Don't try to send data both ways! Another advantage of using iostreams in conjunction with fifos is that you have access to the iostream methods as they would be applied to the fifo. For instance, on line 30 we use the `basic_filebuf()` method `is_open()` to determine whether the fifo is open. If it isn't [Program 11.3b](#) doesn't continue any further. The implementation specifics for [Programs 11.3a](#) and [11.3b](#) are provided in [Program Profiles 11.3a](#) and [11.3b](#).

## Program Profile 11.3b

Program Name

program11-3b.cc

Description

This program reads objects from the fifo using a `ifstream` object. It uses the `ostream_iterator` and the standard transform algorithm to send information through the fifo.

Headers Required

```

<unistd.h>, <iomanip>, <algorithm>, <fstream.h>, <vector>
<iterator>, <sstream.h>, <stdlib.h>, <sys/wait.h>,
<sys/types.h>, <sys/stat.h>, <fcntl.h>, <numeric>

```

Compile and Link Instructions

```

c++ -o program11-3b program11-3b.cc

```

Test Environment

SuSE Linux 7.1, gcc 2.95.2, Solaris 8, Sun Workshop 6.0

Execution Instructions

```

program11.3 & program11-3b

```

Notes

Start [Program 11.3a](#) first. [Program 11.3b](#) has a sleep statement to account for the lack of real synchronization.

### 11.5.3.1 fifo Interface Classes

In addition to simplifying the IPC using `iostreams`, `istream_iterator`, and `ostream_iterator`, we can also simplify matters by encapsulating the `fifo` into a `fifo` class.

**Example 11.23 Declaration of the `fifo` class.**

```
class fifo{
    mutex Mutex;
    //...
protected:
    string Name;
public:
    fifo &operator<<(fifo &In, int X);
    fifo &operator<<(fifo &In, char X);
    fifo &operator>>(fifo &Out, float X);
    //...
};
```

Using this technique, we can easily create `fifos` in the constructor. We can pass them easily as parameters and return values. We can use them in conjunction with the standard container classes. The construction of such a component greatly reduces the amount of code needed to use `fifos`. It provides opportunities for type safety and generally allows the programmer to work at a higher level.

## 11.6 Framework Classes Components for Concurrency

A framework is a class or collection of classes that has a predefined structure and represents a generalized pattern of work. In the same manner that programs provide general solutions to specific problems, frameworks provide specific solutions to classes of problems. That is, an application framework captures the general flow of control for an entire range of programs that all solve or represent problems in a similar fashion. Put another way, an application framework represents a single solution to a family of problems. Frameworks are generic mini self-contained applications. The framework serves as a blueprint for the mini-application. It embodies the fundamental structure or skeleton that the application will have without providing the application details. The framework class specifies the relationships, responsibilities, patterns of work, and protocols between software parts in an object-oriented architecture without providing the implementation details. For instance, we can design a language processor class that captures the general pattern of work for an entire range of applications. The specific pattern of work that the language processor captures is the work involved in taking some input language and translating that language to some output form. This framework consists of a few common software parts:

- Validation components
- Tokenizer components
- Parser components
- Syntax analysis components
- Lexical analysis components

These software parts can be combined to form a very familiar pattern of work:

**Example 11.24 Declarations for the language\_processor class and definition for the process\_input method.**

```
class language_processor {
    //...
protected:
    virtual bool getString(void) = 0;
    virtual bool validateString(void) = 0;
    virtual bool parseString(void) = 0;
    //...
public:
    bool process_input(void);
};

bool language_processor::process_input(void)
{
    getString();
    validateString();
    parseString();
    //...
    compareTokens();
    //...
}
```

First, the language\_processor class is an abstract base class because it contains pure virtual functions:

```
virtual bool getString(void) = 0;
virtual bool validateString(void) = 0;
virtual bool parseString(void) = 0;
```

This means it is not meant to be used directly. It serves as a blueprint for derived classes. The other important thing to note is the process\_input() method. This method captures the general pattern of work that the language\_processor class is meant to generalize. In many ways this is what distinguishes framework classes from other types of classes. The framework not only contains generalized structure and relationships between components, it also captures predefined patterns of work and sequences of action. It provides the skeleton for the pattern of work without providing the implementation details. In this case, the pattern of work is specified by a set of pure virtual functions. So the framework class does not specify how these things are to be done—it only specifies that they should be done and they should be done in a certain order. The derived class has to provide the implementations for the pure virtual functions. The framework class emphasizes the responsibilities of the derived class. Framework classes by definition are contract classes. They require two parties in order to work properly. The framework class does its part but the derived class must provide the implementation details for the pure virtual functions. The commonly found sequence of actions performed by the process\_input() method are found in:

Compilers	Command interpreters
Natural language processors	Encryption/decryption routines
Compression/decompression	File transfer protocols
Graphical user interfaces	Device control, etc.

So by properly designing the `language_processor` class, the pattern of work for an entire range of applications is captured. If the sequence can be properly recorded and tested and debugged, then a wide range of applications can be developed faster by reusing the `language_processor` framework class.

The notion of a framework class is also useful in developing applications that have concurrency requirements. Specifically, the use of agent frameworks and blackboard frameworks captures the basic structure of concurrency and patterns of work within those structures. Michael Wooldridge, in *Reasoning About Rational Agents*, gives us a generalized agent control loop:

Algorithm: Agent control Loop

```
B = B0
while true do
  get next percept p
  B = brf(B,p)
  I = deliberate(B)
  Π = plan(B,I)
  execute(Π)
end while
```

This pattern of work is performed by a wide range of rational agents. If you are developing a program that uses rational agents, then the chances are good that this sequence of actions will be found in your program. This is exactly the type of sequence of action that frameworks are good at capturing. For the agent control loop, the `brf()`, `deliberate()`, `plan()` functions will be pure abstract virtual. The agent control loop specifies what order these functions should be called and how they should be called, and the fact that they should be called. However, what the functions actually do will be determined by a derived class. Once this agent control loop is properly defined, then an entire class of problems has been solved. It turns out that systems consisting of multiple agents executing concurrently are becoming a standard for implementing parallel programming applications. These systems are often called multiagent systems. We discuss agent-oriented architectures in [Chapter 12](#). It is important to note that agent framework classes help to reduce the complexity of developing multiagent systems and multiagent systems are becoming the preferred architecture for implementing medium- to large-scale applications that require concurrency or massive parallelism.

In addition to providing the pattern of work that will be useful for parallel or distributed systems, the framework class can capture the structure with respect to synchronization components such as object-oriented mutexes, semaphores, and message streams. The blackboard structure is a useful medium for multiple agents to communicate through. The blackboard will be a critical region because multiple agents will be able to read and write to it simultaneously. Therefore, the framework class should provide the basic structure for the relationships between the agents, the synchronization components, and the blackboard. For instance, [Example 11.25](#) contains two methods that the framework could use to access the blackboard.

**Example 11.25 Definition of `recordMessage()` and `getMessage()` methods for the `agent_framework` class.**

```
int agent_framework::recordMessage(void)
{
    Mutex.lock();
    BlackBoardStream << Agent[N].message();
    Mutex.unlock();
}
```



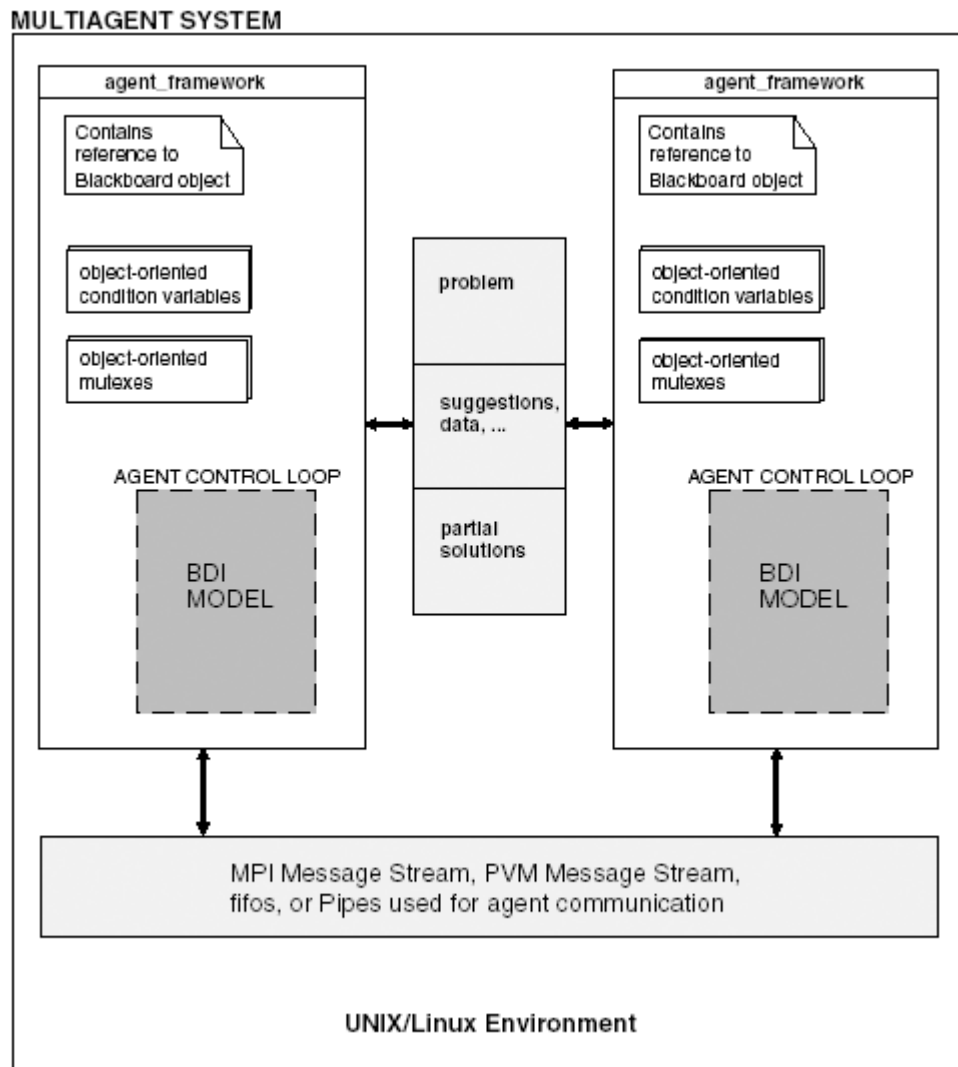
```

int agent_framework::getMessage(void)
{
    Mutex.lock();
    BlackBoardStream >> Values;
    Agent[N].perceive(Values);
    Mutex.unlock();
}

```

Here the framework class will protect the access to the blackboard by using synchronization objects. So when agents read messages from or write messages to the blackboard, the synchronization is already provided by the framework. The programmer does not have to worry about synchronizing blackboard access. [Figure 11-11](#) contains the basic structure of our agent framework and how the framework relates to a blackboard.

**Figure 11-11. The basic structure of the agent\_framework class and how it relates to the blackboard.**



Notice that the framework encapsulates the object-oriented mutexes and condition variables. The agent framework in [Figure 11-11](#) will use either MPI or PVM message streams to communicate in a MPI- or PVM-based system. Recall these message streams were designed as interface classes and allow the programmer to use the iostream metaphor to access the PVM or MPI class. If MPI or PVM are not used, the agents can communicate using sockets, pipes, or even shared memory. In either case, we

recommend that the synchronization primitives be implemented using interface classes since that will make them simpler to use. The blackboard in [Figure 11-11](#) is object-oriented and takes advantage of the genericity provided by template classes. This also simplifies the concurrency requirements. The agents executing concurrently provide an effective model for parallel and distributed programming.

## Summary

The challenges to parallel programming introduced in [Chapter 2](#) can be reasonably approached using the building blocks that we introduced in this chapter. The importance of the interface class in simplifying the use of function libraries cannot be overstated. The interface class introduces consistency of API by wrapping the function calls of libraries such as MPI or PVM. Type safety and reuse is introduced through interface classes. The interface class allows the programmer to work with a familiar metaphor, as in the case of PVM streams or MPI streams. IPC is simplified by connecting pipe or message streams to iostreams and overriding the << inserter and >> extractor operators for user-defined classes. The `ostream_iterator` class proves to be very useful in sending entire container objects and their contents between processes. The `ostream_iterator` and `istream_iterator` also provide the glue between the standard algorithms and IPC components and techniques. Since a large number of parallel or distributed applications use the message-passing model, any technique that simplifies passing various datatypes between processes will simplify the programming required for the application. Using iostreams, the `ostream_iterator` and `istream_iterator` does this simplification. The framework class is introduced here as the basic building block of concurrency applications. We consider classes like the mutex classes, condition variable classes, and the stream classes to low-level components that should be hidden from the programmer within the framework class (where possible!). When building medium- to large-scale applications that require concurrency, the programmer should not have to focus on these low-level components. Ideally, the framework will be the base-level building block for concurrency approaches, which we introduce in the remainder of this book. The framework will provide us with patterns for peer-to-peer and client-server interaction. We can have numeric frameworks, database frameworks, agent frameworks, blackboard frameworks, GUI frameworks, and so on. The approach that we advocate for implementing concurrency requirements builds applications from a collection of frameworks that already have the proper synchronization components wired into the proper relationships. In [Chapters 12](#) and [13](#), we take a closer look at frameworks that support concurrency. We also introduce the use of standard C++ algorithms, containers, and function objects to manage the creation and spawning of multiple tasks or threads in applications that require concurrency.

## Chapter 12. Implementing Agent-Oriented Architectures

Much remains to be done before we understand how people construct their problem representations and the role those representations play in problem solving. But we know enough already to suggest that the representations people use—both propositional and pictorial—can be simulated by computers.

—Herbert A. Simon, *Machine as Mind* (Android Epistemology)

In this Chapter

- [What are Agents?](#)
- [What is Agent-Oriented Programming?](#)
- [Basic Agent Components](#)
- [Implementing Agents in C++](#)
- [Multiagent Systems](#)
- [Summary](#)

If sequential (procedure-based) programming solutions were practical in every situation, then there would be no need for parallel or distributed programming techniques. In many situations, sequential programming techniques are simply inadequate for the demands and the sophistication of today's computer users. As developers scramble for new approaches to meet the growing challenges that user requirements present, alternative software models are created. Better ways to organize and think about how software should be constructed are discovered. Structured programming was presented as an improvement over procedureless goto/jump-filled programming. Object-oriented programming was presented as an improvement over structured programming. In many ways agents and agent-oriented programming is an improvement over object-oriented programming. Agents present yet another (more sophisticated) method for organizing and thinking about distributed/parallel programs.

### 12.1 What are Agents?

There was a lot of controversy over what constituted an object when object programming was initially introduced. There is a similar controversy over exactly what constitutes an agent. Many proponents define agents as autonomous, continuously executing programs that act on behalf of a user. However, this definition can be applied to some UNIX daemons, or even some device drivers. Others add the requirements that the agent must have special knowledge of the user, must execute in an environment inhabited by other agents, and must function only within the specified environment. These requirements would exclude other programs considered to be agents by some. For instance, many e-mail agents act alone and may function in multiple environments. In addition to agent requirements, various groups in the agent community have introduced terms like softbot, knowbot, software broker, and smart object to describe agents. We define the term agent iteratively in this chapter. We start with some simple, agreed-upon partial definitions and construct a definition that is practical for C++ programmers.

One commonly found definition defines an agent as an entity that functions continuously and autonomously in an environment in which other processes take place and other agents exist. Although it is tempting to accept this definition and move on, we cannot because it too easily describes other kinds of software constructions. Many object-oriented components function continuously and autonomously in an environment in which other processes take place and other objects exist. In fact, many CORBA-based client-server systems fit this description! So if we exchange the word object for agent in this definition it accurately describes many object-oriented systems. A look at a more formal source, the

Foundation for Intelligent Physical Agents (FIPA) specification defines the term agent accordingly:

An Agent is the fundamental actor in a domain. It combines one or more service capabilities into a unified and integrated execution model which can include access to external software, human users and communication facilities.

While this definition has a more structured feel, it also needs further clarification because many servers (some object-oriented and some not) fit this definition. This definition as is would include too many types of programs and software constructs to be useful. Although we rely on the FIPA specification where we discuss agents, this basic definition requires further work.

### 12.1.1 Agents: A First-Cut Definition

One of the reasons that the word object can in so many instances replace the word agent in so many definitions and descriptions of agent is because agents are inherently based on objects. In fact, our initial requirement on the definition for an agent is that it first fit the definition of an object,[\[1\]](#) that is, we designate an agent as a certain kind of object. This chapter is largely about what makes an agent different from other classes of objects. In the same sense that C++ has support for interface classes, container classes, and framework classes, we can also designate agent classes. This brings us to our second requirement on the definition of agents within a C++ environment. In C++ an agent is implemented using the notion of a class. The different types of classes are distinguished by how they function or how they are structured. For instance, a container class describes an object used to hold or contain other objects. An interface class is used to describe an object that transforms or adapts the interface of another object. A framework class describes an object that contains a pattern of work that is common to a family of other objects. An agent class will be used to define objects that have what Yohav Shoham (1997) describes as a mental state: "The mental state will contain such components as beliefs, capabilities, choices, and commitments." This mental state is often partially summarized by the Belief Desires and Intentions (BDI) model. We extend the BDI model to include actions. So in our first-cut definition of agents, we define an agent as a piece of software meeting the following three requirements:

<sup>[1]</sup> When we use the term object in the definition of agent we include its AI cousins: actor and frame.

1. A certain type of object (not all objects are agents)
2. Implemented using the notion of a class (encapsulation, inheritance, and polymorphism are important for agents!)
3. Contains a set of behaviors and attributes that must include beliefs, desires, intentions, and actions

For our purposes, agents are by definition rational software components. Before we define agents further, let's look at the types of agents that are commonly implemented.

### 12.1.2 Types of Agents

Several categories of agents have emerged. Although not every agent fits into one of these categories, the categories are generally descriptive of the majority of agents in practical use. [Table 12-1](#) contains five major categories of agents. Obviously there are hybrid agents that fit into more than one category at the same time. There are no hard and fast rules that determine which agents fit into any particular category. These categories are presented for convenience and as a starting point when trying to classify agents that you may have to develop or work with.

**Table 12-1. Five Major Categories of Agents**

<b>Agent Categories</b>	<b>Description</b>
Interface agents	Represent the next generation of human–computer interaction. These agents provide new user interfaces to the computer.
Search agents	Perform various types of information retrieval.
Monitor/control agents	Patrol, observe, audit, manage, and monitor devices and conditions, data, and processes.
Acquisition agents	Authorized to acquire some good or service on behalf of the user.
Decision support agents	Provide analysis, information synthesis, condition and data interpretation, planning, and evaluation.

[Table 12-1](#) represents a functional breakdown of the agent categories. It does not specify any particular components that the agents must have. It only specifies the types of activities that the agents engage in. In fact, these categories are not the exclusive domain of agents. Other classes of software such as expert systems and object-oriented systems have categorizations very similar to these. In some cases, the only difference is that we are talking about agents as opposed to objects or expert systems.

### 12.1.3 What is the Difference between Objects and Agents?

One of the fundamental requirements for an agent is that it first meets the conditions of object orientation. So agents and objects have more things in common than many of the agent proponents would like to admit. It is the function and construction of the object that places it into the agent column. Objects are by definition self-contained and exhibit a certain amount of autonomy. Once the degree of autonomy crosses a certain threshold and the object is given cognitive data structures such as those found in the BDI model, then the object is an agent. An autonomous rational object is an agent. [\[2\]](#) An object is considered rational when it has:

<sup>[2]</sup> We intentionally avoid the term intelligent. It is not currently known whether we will ever produce intelligent software. However, we can undoubtedly produce rational software based on well understood, logical formalisms.

- Methods that implement some form of deduction, induction, or abduction
- Data members that are implementations of cognitive data structures

Keep in mind that in object-oriented programming, the routines defined for a class are called methods and in C++ they are called member functions. The variables or data components defined for a class are called attributes and in C++ they are called data members. If some of the member functions are used to perform deduction, induction, or abduction on the data members that are implementations of cognitive data structures, then the object is rational. If the rational object also crosses a certain threshold of autonomy, then it is an agent.

Cognitive data structures are structures used to represent mental constructs like beliefs, intentions, commitments, decisions, moods, and knowledge. For instance, we could designate a believe structure

using a C++ set:

```
set<statements> Beliefs;  
  
struct statement {  
//...  
    float ArrivalTime;  
    float DepartureTime;  
    string Destination;  
    //...  
};
```

where the statements are about schedule some form of public transportation. A collection of these statements is stored within `set<statements>` and represents the agent's beliefs. This is what we mean by data members that are implementations of cognitive data structures. The agent would declare the data member accordingly:

## S 12.1. Deduction, Induction, and Abduction

Deduction, induction, and abduction are processes used to draw conclusions from a set of statements or a collection of data. The process of deduction allows the reasoner to deduce a conclusion from a set of statements. If the statements are true and the reasoner follows the proper rules of inference, then the conclusion is said to be necessarily true or that it follows by necessity. For instance:

All three-sided figures are triangles.

This is a three-sided figure.

This is a triangle. ← Conclusion arrived at by deduction

The rules of inference are guidelines and restrictions that determine how the reasoner may move from one statement to another. The rules of inference determine when statements are logically equivalent and the conditions under which one statement may be transformed into another. [Sidebar Figure 12-1](#) contains the eight most basic rules of inference.

. **Sidebar Figure 12-1 Rules of inference taken from the back inside cover of COPI symbolic reasoning.**

1. Modus Ponens

$$\begin{array}{l} p \supset q \\ p \\ \hline \therefore q \end{array}$$

2. Modus Tollens

$$\begin{array}{l} p \supset q \\ \neg q \\ \hline \therefore \neg p \end{array}$$

3. Hypothetical Syllogisms

$$\begin{array}{l} p \supset q \\ q \supset r \\ \hline \therefore p \supset r \end{array}$$

4. Disjunctive Syllogism

$$\begin{array}{l} p \supset q \\ \neg p \\ \hline \therefore q \end{array}$$

5. Constructive Dilemma

$$\begin{array}{l} (p \supset q) \bullet (r \supset s) \\ \hline \therefore q \vee s \end{array}$$

6. Absorption

$$\begin{array}{l} p \supset q \\ \hline \therefore p \supset (p \bullet q) \end{array}$$

7. Simplification

$$\begin{array}{l} p \bullet q \\ \hline \therefore p \end{array}$$

8. Conjunction

$$\begin{array}{l} p \\ q \\ \hline \therefore p \bullet q \end{array}$$

9. Addition

$$\begin{array}{l} p \\ \hline \therefore p \vee q \end{array}$$

The process of induction allows the reasoner to induce the conclusion from a set of statements taken to be facts. For instance:

It rained yesterday.

It rained the day before.

It rained all last week.

It will rain tomorrow. ← Conclusion arrived at by induction

Whereas the conclusions in the deductive process are said to be necessarily true (if the rules of inference were applied correctly), the conclusions in an inductive process have only a probability of being true. How close that probability is to 100% will depend on the nature and context of the statements and data they are drawn from. The process of abduction allows the reasoner to draw the most plausible conclusion based on a set of statements or data. For instance:

Articles of the defendant's clothing were found at the scene of the crime.

The defendant and the deceased recently had a violent argument.

The defendant's DNA was found at the scene of the crime.

The defendant is the perpetrator of the crime ← Conclusion arrived at by abduction

Deduction, induction, and abduction are the three fundamental processes found in logic. They provide for logic what calculation and arithmetic provide for mathematics. The ability to correctly move from premises (statements, data, and facts) to conclusions is the process that we call reasoning.

```
class agent {  
    //...  
    set<statements> Beliefs;  
    //...  
};
```

The agent class uses deduction, induction, or abduction to process its Beliefs in order to form intentions, commitments, or plans. A closer look at our definition of agents states that if it is a rational autonomous object, it is an agent. If it is not rational, then it's not an agent, it's just an object. The degree of autonomy is another area of debate and we will examine it closely later in this chapter.

## 12.2 What is Agent-Oriented Programming?

Agent-oriented programming is the process of assigning the work a program has to one or more agents. The WBS (Work Breakdown Structure) consists only of agents. If all the work that a program does can be assigned to one or more agents, then it is a pure agent-oriented program and all of the design and development involved only requires agent-oriented programming. In many situations, agents will be involved with other kinds of objects and systems that are not agent-oriented and therefore the entire programming effort is not called agent-oriented programming, which is often the case when agents are involved with database servers, application servers, and other types of object-oriented systems. Whether producing software systems that are completely agent-oriented or only partially agent-oriented, agent-oriented programming produces rational object-oriented software components.

### 12.2.1 Why Agents Work for Distributed Programming

Practical distributed programs rise out of necessity. Typically, there is some resource located on another computer or network separated from the program that needs it. These resources often take the form of databases, Web servers, e-mail servers, application servers, printers, and large storage devices. The resources are usually managed by a piece of software called a server. The software that needs access to the resources is referred to a client. The fact that the resources are located on different computers than the client leads to distributed architectures. In most cases, it does not make sense to attempt to combine these programs into one large program that runs on a single computer and in a single address space. Furthermore, there are many programs developed at different times, by different developers, and for different purposes that end up taking advantage of each other's services. The application that uses these programs evolved over time and the result was a distributed application. Since these programs are separate, they will each have their own address space and resources. When these programs are used together to form a single application or collectively solve a program they form a distributed program. It turns out that the distributed program architecture provides very flexible architectures that can be used for large-scale applications. In so many practical applications, the requirements for distribution are discovered after the fact. However, good software engineering and design techniques can be used to identify when applications should be distributed. Once you know that you need to develop a distributed application, the next question is how it should be distributed. What model should be used? Although there are very many different client-server and peer-to-peer models available in this book, we focus on only two: blackboard architectures and multiagent architectures.

Both of these architectures can take advantage of agents because agents are inherently self-contained,



autonomous, and rational software structures. Because agents are rational it means they know their purpose. Regular objects have a purpose and agents know what that purpose is. Identifying the purpose of each aspect of a piece of software is a natural process. It is straightforward to recognize the purpose of a piece of software during the design phase. Assigning that purpose to an agent is an easy form of software decomposition. The WBS becomes a matter of which class of agent to delegate the work to. Since the agent is the unit of modularity in an AOP (agent-oriented program), the work of distribution is reduced to finding means for multiple agents to communicate. The process of designing the original agent class hides the effort required for identifying the WBS of a distributed program. Once we get over the hurdles that agents are really rational objects, we can then take advantage of the CORBA specification to design truly distributed multiagent systems. CORBA hides the complexity of distributed programming and communication over networks, intranets, and the Internet. [Chapter 8](#) contains a simple overview of distributed programming using CORBA. Since agents are objects, the entire discussion of CORBA is applicable. [Chapter 6](#) introduces the PVM (Parallel Virtual Machine). PVM can also be used to greatly simplify the communication between agents executing on different processes or on different computers. Agents can be implemented as CORBA objects, or they can be assigned to separate PVM processes. In both cases, the communication is simple and straightforward. When two or more agents are involved within a single application they are multiagent systems. The agents may still use CORBA, PVM, or the MPI (Message Passing Interface) to communicate if they are on the same computer. Agents within different processes may also use traditional methods of IPC such as fifos, shared memory, and pipes to communicate. There are three fundamental challenges in distributed programming:

1. Identifying the WBS of the distributed solution
2. Implementing effective and efficient communication between the distributed components
3. Dealing with exceptions, errors, and partial failures

While there is nothing inherent within the notion of an agent class to deal with item 2, items 1 and 3 are almost implicit in the agent design itself. Each agent's rationality defines its purpose and thereby the part that it is to play in the software solution. Since agents are self-contained and autonomous, a good agent class design will include the necessary fault tolerance.

### **12.2.2 Agents and Parallel Programming**

Agents deployed in an environment where multiple processors or concurrently executing threads offer the same advantages as they do in distributed programming, with the addition that cooperation between agents is much easier to program. The PVM and MPI environments can also be used for message passing between agents that are collectively solving some kind of problem. Again, the rationality of the agents makes it easier to understand to design the WBS for parallelism. The common obstacles encountered in parallel programming are:

1. Dividing the work effectively and efficiently between two or more software components
2. Coordinating the concurrently executing software components
3. Designing appropriate communication (where needed) between the components
4. Dealing with exceptions, errors, and partial failures (if the agents are on separate computers)

Multiagent parallel architectures tend to be loosely coupled, that is, the communication and the interdependency is minimal. Each agent knows its purpose and has methods to accomplish its purpose. Whereas obstacle 3 is not inherently dealt with by the agent class, obstacles 1, 2, and 4 are easily managed by the implicit capabilities of the agent classes. For example, the impact of obstacle 2 is reduced because each agent is rational, has a purpose, and has the ways and means to accomplish its purpose. So the responsibility is shifted away from some coordination and control algorithm to the

actions of each agent. The impact of obstacle 4 is reduced because agents are self-contained, rational, and autonomous and a good class design will include the necessary fault tolerance. Since the agent's state is encapsulated, the responsibility to protect critical sections within the agent object is the responsibility of the agent class. The agent will enforce its own data access policies. [Table 12-2](#) shows the state access policies from which agents can choose.

**Table 12-2. State Access Policies**

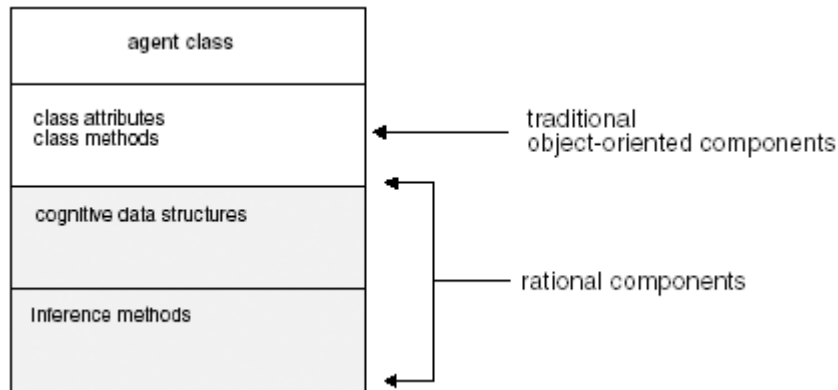
<b>Read-Write Algorithm Types</b>	<b>Meaning</b>
EREW	Exclusive Read Exclusive Write
CREW	Concurrent Read Exclusive Write
ERCW	Exclusive Read Concurrent Write
CRCW	Concurrent Read Concurrent Write

Each agent's class will determine which access policy is acceptable in a multiagent environment. In some cases, combinations of the access policies in [Table 12-2](#) are implemented. This makes the parallel programming easier. The developer can work at a higher level without having to worry about mutexes, semaphores, and so on. Multiagent solutions allow the developer to work at a higher level without getting bogged down with the minutia of coordinating every function call and data access. Each agent has a purpose. Each agent is rational and therefore has a logic for achieving its purpose. The programming process looks a lot more like task delegation as opposed to the typical task coordination paradigms for traditional parallel programming. Since agent-oriented programming is a specific kind of object-oriented programming, agents use a more declarative mode of parallel programming than the traditional procedural-based programming that is often implemented in languages such as Fortran or C. The developer specifies what needs to be done and which agents should do it and the parallelism almost takes care of itself. There is always some amount of coordination and communication programming required, but agent-oriented programming keeps it to a minimum. However, all these advantages depend on the existence of agent classes. Someone has to design and code the agent classes. Let's now look at what an agent class will contain.

## 12.3 Basic Agent Components

An agent is declared using the class keyword. The components of an agent will consist of C++ data members and member functions. [Figure 12-1](#) shows the logical layout of an agent class.

**Figure 12-1. Logical layout of an agent class.**



The class attributes and methods in [Figure 12-1](#) refer to the typical initialization, read, and write methods that any object would have. The attributes would include state variables that define the object. The methods would include constructors, destructors, assignment operators, exception handlers, and so on. If we stopped with these attributes and methods we would have only a traditional object. The cognitive data structures and the inference methods make up the rational component. It is the rational component that transforms an object into an agent.

### 12.3.1 Cognitive Data Structures

A data structure is a set of rules for logically organizing data and the rules for accessing that logical organization. It is a method of organization that specifies both how the data should be conceptually structured and how access operations are allowed to be applied to that structure. Whereas datatypes and ADTs (abstract datatypes) focus on what, data structures focus on how. For example, the integer datatype specifies an entity that has a data component and a number of arithmetic operations (i.e., addition, subtraction, multiplication, division, modulo, etc.). That data component does not have a fractional part. The data component consists of negative and positive numbers, and so on. The datatype specification does not mention how the integers should be used or accessed. On the other hand, a data structure specification such as a stack specifies a list of elements stored in a LIFO (last-in-first-out) order. The stack data structure also specifies that elements may only be taken out one at a time from the top of the stack, that is, the last element inserted must be removed before any other elements can be accessed. So not only does the stack data structure specify how the elements are organized, it also specifies how the elements are accessed (i.e., visited, read, changed, deleted, etc.). Cognitive data structures restrict the rules for organizing and accessing data to those found in the fields of logic and epistemology. The rules of inference, the methods of inference (i.e., deduction, induction, and abduction), the notions of epistemic data, knowledge, justification, belief, premises, propositions, fallacies, and conclusions are the defining features of cognitive data structures.<sup>[3]</sup> Whereas algorithms for sorting, searching, and iterating are commonplace for traditional data structures, inference methods are more commonplace for cognitive data structures. The ADTs used with cognitive data structures often include:

<sup>[3]</sup> Mentalistic concepts such as imagination, paranoia, anxiety, happiness, sadness, and so on are intentionally excluded from our definition of cognitive data structures. Our focus is on rational epistemic software, not intelligent software.

Questions	Events
Facts	Time
Propositions	Fallacies
Beliefs	Purpose
Statements	Justification
Conclusions	

Of course, other datatypes can be used with a cognitive data structure but these are characteristic of programs that use rational software components such as agents. These ADTs are normally implemented as datatypes using the struct or class keywords. For instance:

```
struct question{
    //...
    string RequiredInformation;
    target_object QuestionDomain;
    string Tense;
    string Mood;
    //...
};

class justification{
    //...
    time EventTime;
    bool Observed;
    bool Present;
    //...
};
```

The C++ template and container classes can be used to organize cognitive data structures such as knowledge. For instance:

```
class preliminary_knowledge{
    //...
    map<question,belief> Opinion;
    map<conclusion,justification> SimpleKnowledge;
    set<propositions> Argument;
    //...
};
```

### 12.3.1.2 Inference Methods

The inference methods in [Figure 12-1](#) refer to deduction, induction, and abduction. (See [Sidebar 12.1](#) for an explanation of these methods.) While inference methods are called for in an agent architecture, there is no specific mention on how the inference methods are implemented. Deduction, induction, and abduction are high-level processes. The details of how to implement these processes are up to the software developer. Inference is the process of deriving logical conclusions from premises known or

assumed to be true. There is no one right way to implement an inferencing process, sometimes called an inference engine. However, there are several commonly used methods for implementing inference. Forward-chaining or backward-chaining techniques can be used. Means–end analysis techniques can be used. Graph traversal algorithms such as DFS (Depth First Search) and BFS (Breadth First Search) can also be used. There is a host of theorem-proving techniques that can be used to implement inference methods and inference engines. The important point to note here is that an agent class will have one or more inference methods. [Table 12-3](#) contains descriptions for the most basic techniques used to implement the inference methods.

**Table 12-3. Tables for Descriptions of the Most Basic Inference Implementation Techniques**

**Inference Implementation Techniques    Description**

Backward Chaining	Purpose- or goal-driven approach in which a process starts from a proposition, statement, or hypothesis and searches for supporting evidence.
Forward Chaining	Data-driven approach that starts from available data or facts and moves toward conclusions.
Means-end Analysis	Uses a set of operators to solve one subproblem at a time until the entire problem is solved (opportunistic).

These techniques are well understood and widely available in many libraries, frameworks, and some programming languages. These techniques are the building blocks for the basic inference methods. To see how this inferencing works lets use one of the rules of inference Modus Ponens and build a simple inference method to support it. Take the following statement: If there is a bus trip from Detroit to New York then John will go on vacation. If we establish that there is a bus trip from Detroit to New York, then we know that John will go on vacation. The Modus Ponens form of this is:

$P \rightarrow Q$

P

\_\_\_\_\_

Q

where:

P = If there is a bus trip from Detroit to New York

Q = John will go on vacation

We could design a simple decision support agent to tell us whether John will go on vacation or not. That agent would need to know something about possible bus trips. Let's say we have a list of bus trips:

Toledo to Cleveland	Detroit to Chicago	Youngstown to New York
Cleveland to Columbus	Cincinnati to Detroit	Detroit to Toledo

Columbus to New York

Cincinnati to Youngstown

Each of these trips represents commitments by the ABC Bus Company. If our agent has access to the ABC Bus Company's schedule, then this list of trips can be used to represent part of our agent's beliefs. The question is, how do we get from a list of trips to beliefs? First, let's design a simple statement structure.

```
struct existing_trip{
    //...
    string From;
    time Departure;
    string To;
    time Arrival;
    //...
};
```

Next let's use a container class that will represent our agent's beliefs about bus trips.

```
set<existing_trip> BusTripKnowledge;
```

If the bus trip is contained in the set BusTripKnowledge, then our agent believes that the bus trip will take place from a certain origin to a certain destination at a certain time. So we might construct a trip accordingly:

```
//...
existing_trip Trip;
Trip.From.append("Toledo");
Trip.To.append("Cleveland");
Trip.Departure("4:30");
Trip.Arrival("5:45");
BusTripKnowledge.insert(Trip);
//...
```

If we place each trip into the BusTripKnowledge set, then our agent's beliefs about the ABC Bus Company's trips are complete. Notice that there is no single trip from Detroit to New York. However, John can get to New York from Detroit if he takes the following bus trips:

Detroit to Toledo

Toledo to Cleveland

Cleveland to Columbus

Columbus to New York

So while the ABC Bus Company does not provide a one-stop trip, it does provide a multistop trip. The problem is how does our agent know this? The agent needs some way to conclude based on what it knows about bus trips that there is a trip from Detroit to New York. We use a simple chaining process. We search our BusTripKnowledge for the first trip leaving from Detroit. We find one: Detroit to Chicago. We check the To attribute of this trip. If it is equal to "New York", we stop because we have found a trip. If it is not, we save this trip on a stack. We then search through the trips to see if there is another trip whose From attribute = "Chicago". We find that there are no buses leaving from Chicago. Therefore, we pop the Detroit to Chicago trip from the stack. We note that we have used this trip, and we search for the next trip leaving from Detroit to anywhere. We find a trip: Detroit to Toledo. We check to see if the To attribute = "New York", and since it does not, we save this trip on a stack. We then search through the trips to see if there is another trip whose From attribute = "Toledo". Here we find one: Toledo to Cleveland. We then place this trip on the stack. We then search through the trips to

see if there is another trip whose From attribute = "Cleveland". At each trip we check the To attribute. If the To attribute = "New York", then the trips on the stack represent a bus trip from Detroit to New York, with the starting trip at the bottom of the stack and the ending trip on the top of the stack. If we go through the entire list and none of the To attributes = "New York", or we run out of To attributes for the trip on top of the stack, then we pop the top element of the stack and search for the next item whose From attribute is equal to the To attribute of the element on the top of the stack. This process is repeated until either the stack is empty or we've found a trip. This process uses a simplified DFS technique to determine if there is a trip from point A to point B.

Our simple agent would use this DFS technique to establish the existence of the trip from Detroit to New York. Once this fact was established, then the agent would update its beliefs about John. The agent will now believe that John is going on vacation. Let's say we added another precondition to John's vacation:

If John adds 15 or more new clients, then his profits > 150000.

If John's profits are > 150000 and there is a bus trip from Detroit to New York, then John will go on vacation.

Here the agent must establish whether John's profits are > 150000 and whether there is a bus trip from Detroit to New York. To establish whether John's profits are > 150000, the agent must first establish whether John has added 15 or more new clients. Suppose we convince the software agent that John has added 23 new clients. Then the agent can infer that John's profits are > 150000. From BusTripKnowledge the agent was able to infer that there was a trip from Detroit to New York. Using the beliefs about the bus trips, and the belief about the 23 new clients, the agent uses the process of forward chaining to deduce that John will go on vacation. We call this forward chaining from the fact that with 23 new clients added and the bus schedule facts, the agent moved to the conclusion. The inference form of this process looks like:

A  $\rightarrow$  B

(B and C)  $\rightarrow$  D

A

C

---

D

where

A = If John adds 15 or more new clients

B = Profits > 150000

C = There is a bus trip from Detroit to New York

D = John will go on vacation

In this example, the agent believes A and C to be true. Using the rules of inference, B and D are established to be true. Therefore, the agent will commit to tell us that John will go on vacation. This kind of processing could be assigned to an agent in a situation where a manager has hundreds or even thousands of employees and decides to have agent software regularly schedule employee hours. The manager would then consult the agent to see who was working, who was on sick leave, who was on vacation, and so on. The agent would be given knowledge and authority to assign work schedules. Each week the agent would commit to a number of acceptable work schedules, vacations, and sick leaves. The agent in this case uses simple forward chaining and DFS to make its inferences. To implement this

kind of inferencing we used structs, and stack and set classes. These classes are used to hold knowledge, propositions, and patterns of reasoning. They allow us to implement our CDS (Cognitive Data Structures). We used DFS techniques to move through the stack data structures and set structures to support the process of inference.

The combination of chaining and DFS produces a process whereby one proposition can be affirmed on the basis of previous propositions that have already been accepted. This is important because our agent will know that it is inherently correct when it accomplishes its objectives based on inference. This also affects parallel programming considerations. The fact that the agent is rational and moves according to rules of inference allows the developer to focus on correctly modeling the task that the agent performs instead of getting bogged down in attempts to explicitly control the parallelism in the program. The minimal requirements of parallelization DCS (decomposition, communication, and synchronization) are in large part addressed by the architecture of the agent. Each agent has a rationale for its behavior. That rationale will be based on well defined, well understood rules of inference. The decomposition happens simply as a matter of assigning the agent one or more prime directives. Thinking of the WBS in this way is natural and ultimately results in parallel or distributed programs that are easier to maintain and enhance. It is easier to think about communication between agents than communication between anonymous modules because the boundaries between agents are clear and obvious. Each agent has a purpose that is immediately apparent. The knowledge or information each agent needs to achieve its purpose is easily determined. To allow the agents to communicate, the developer can use simple MPI calls or the object communication facilities that are part of any CORBA implementation. The most challenging aspect of the communication, namely figuring out:

- what needs to be communicated
- who needs to communicate
- when the communication should occur
- what format the communication should be in

is implied in the design of the agents. All that is left is the physical implementation of the communication, which is easily handled by any of the libraries that support parallelism that we have discussed in this book. Finally, the problems of synchronization are reduced because the agent's rationale tells it when it can and should perform an action. Therefore, the complicated issues of synchronization are transformed into simpler issues of cooperation. This subtle difference is an important paradigm shift because it simplifies what the software developer has to think about. Lets look a little closer at the basic layout of an agent and how we can implement it in C++.

## 12.4 Implementing Agents in C++

We will explore a simplified variation of our previous example of an agent and demonstrate how it can be approached in C++. The purpose of this agent is to schedule and book vacations for the owner of the ABC Auto Repair Company. The owner has dozens of employees and therefore doesn't have time to figure out when and where to go on vacation. Furthermore, unless the owner is making a certain amount of profit, vacations are out of the question. So the owner has acquired agent software that will plan and schedule vacations at various times throughout the year if the conditions are right. As far as the owner is concerned, the primary selling feature is that the agent runs unattended. Once the agent is installed on the computer the owner doesn't have to bother with it. When the agent determines that an appropriate time for a vacation has arrived, the agent will schedule the vacation, book the hotel and transportation, and then e-mail the owner an itinerary. The only responsibility the owner has is during the setup of the agent. The owner has to specify where he would like to go and how much profit the business must make before he can go. Let's look at how this agent could be constructed. Recall from [Figure 12-1](#) that the rational component of an agent class consists of cognitive data structures and inference methods.



The cognitive data structures help to capture beliefs, propositions, notions of epistemic data, knowledge, fallacies, facts, and so on. The agent class uses inference methods to access these cognitive data structures in the process of problem solving and task performance. The standard C++ library comes with a set of container classes and algorithms that can be used to implement the CDS and the inference methods.

### 12.4.1 Proposition Datatypes and Belief Structures

This agent has beliefs about the performance of the owners auto repair business. The beliefs capture information about how many customers per hour, the bay utilization per day, and the total sales during the period. Furthermore, the agent knows that the owner only likes bus trips. Therefore, the agent keeps up on any available bus trips that the owner might enjoy. In a math-intensive program, the primary datatypes are integers and floating point numbers. In a graphics-intensive program, the primary datatypes are pixels, lines, colors, circles, and so on. In an agent-oriented program, the primary datatypes are propositions, rules, statements, literals, and strings. We will use the object-oriented support in C++ to build a few datatypes that are native to agent-oriented programming. [Example 12.1](#) shows the declaration for a proposition class.

#### Example 12.1 Declaration of a proposition class.

```
template<class C> class proposition {
//...
protected:
    list<C> UniverseOfDiscourse;
    bool TruthValue;
public:
    virtual bool operator()(void) = 0;
    bool operator&&(proposition &X);
    bool operator||(proposition &X);
    bool operator||(bool X);
    bool operator&&(bool X);
    operator void*();
    bool operator!(void);
    bool possible(proposition &X);
    bool necessary(proposition &X);
    void universe(list<C> &X);
//...
};
```

A proposition is a statement in which the subject is affirmed or denied by the predicate. A proposition is either true or false. A proposition can be used to capture any single belief that the agent has. Also, other information that the agent does not necessarily believe but is offered to the agent will be presented as a proposition. The proposition is a cognitive datatype. It should be just as functional in an agent-oriented program as a floating point or integer datatype in a math-oriented program. Therefore, we use the operator overloading facilities of C++ to provide some of the basic operators that are applicable to propositions. [Table 12-4](#) shows how the operators are mapped to logic operators.

The proposition class in [Example 12.1](#) is a scaled-down version. The goal of this class is to allow the proposition datatype to be used just as easily and naturally as any other C++ datatype. Notice that the proposition class has the declaration:

**Table 12-4. How the Operators are Mapped to Logical Operators**

<b>C++ User-Defined Operators</b>	<b>Commonly Used Logical Operators</b>
&&	$\wedge$
	$\vee$
!	$\sim$
possible	$\blacklozenge$
necessary	$\square$

```
virtual bool operator()(void) = 0;
```

This is called a pure virtual method. When a class has a pure virtual method, it means the class is an abstract class and cannot be directly instantiated. This is because there is no definition for the pure virtual function in the class. The function is only declared, not defined. Abstract classes are used to define policies and blueprints for derived classes. A derived class must define any pure virtual functions that it inherits from the abstract class. Here the proposition class is used to define the minimum capability that any descendant will have. Another important thing to notice about the proposition class in [Example 12.1](#) is it's also a template class. It contains the data member:

```
list<C> UniverseOfDiscourse;
```

This data member will be used to hold the universe of discourse that the proposition belongs to. In logic, the universe of discourse contains all of the legal things that may be considered during a discussion. Here, we use a list container. Since the topics under consideration in a universe of discourse can take on different types, we use a container class. We make the UniverseOfDiscourse protected instead of private so that it can be accessed by all descendants of the proposition class. The proposition class also has the capability to deal with logical necessity and possibility, the major themes in modal logic that are also useful in agent-oriented programming. Modal logic allows the agent to deal with what is possibly true or what is necessarily true. [Table 12-4](#) lists the primary operators used for logical possibility and necessity. We provide these methods for exposition purposes only; the implementations of these methods are beyond the scope of this book. However, they are part of the proposition classes that the authors use in practice. To make the proposition class usable, we derive a new class that we name trip\_announcement. The trip\_announcement class is a statement about the existence of a bus trip from some point of origin to some destination. For instance: There is a bus trip from Detroit to Toledo. This makes a statement that is either true or false. If we were concerned with when this statement was true or false we might imply temporal logic. Temporal logic deals with the logic of time. Agents also employ temporal reasoning. But here all propositions refer to the current time. This statement asserts that there is currently a bus trip from Detroit to Toledo. The agent will either be able to verify this and therefore believe or reject it as a false statement. [Example 12.2](#) shows the declaration of the trip\_announcement class.

### Example 12.2 Declaration of the trip\_announcement class.

```
class trip_announcement : public proposition<trip_announcement>{
//...
protected:
    string Origin;
    string Destination;
    stack<trip_announcement> Candidates;
public:
    bool operator()(void);
    bool operator==(const trip_announcement &X) const;
    void origin(string X);
    string origin(void);
    void destination(string X);
    string destination(void);
    bool directTrip(void);
    bool validTrip(list<trip_announcement>::iterator I,
                  string TempOrigin);
    stack<trip_announcement> candidates(void);
    friend bool operator||(bool X,trip_announcement &Y);
    friend bool operator&&(bool X,trip_announcement &Y);
//...
};
```

Notice that the trip\_announcement class inherits the proposition class. Recall that the proposition class is a template class and requires a parameter to determine its type. The declaration:

```
class trip_announcement : public proposition<trip_announcement>
    {...};
```

provides the proposition class with a type. It is also important to note that the trip\_announcement class defines the operator(). Therefore, trip\_announcement is a concrete class as opposed to an abstract class. We may declare and use the trip\_announcement proposition directly within our agent program. The trip\_announcement class adds some additional data members:

Origin  
Destination  
Candidates

These data members are used to contain the origin and destination of a bus trip. If the bus trip requires transfers from one bus to another and multiple layovers, then the Candidates data member will contain the complete route involved. Therefore, the trip\_announcement object will be a statement about a bus trip and the route involved. The trip\_announcement class also defines some additional operators. These operators help to put the trip\_announcement class on equal footing with built-in C++ datatypes. In addition to beliefs about trips, the agent also has beliefs about the performance of areas within the owner's business. These beliefs differ in structure but are still basically statements that will be either true or false. So, we again use the proposition class as a base class. [Example 12.3](#) shows the declaration for the performance\_statement class.

**Example 12.3 Declaration for the performance\_statement class.**

```
class performance_statement : public proposition<performance_
                                statement>{
    //...
    int Bays;
    float Sales;
    float PerHour;
public:
    bool operator() (void);
    bool operator==(const performance_statement &X) const;
    void bays(int X);
    int bays(void);
    float sales(void);
    void sales(float X);
    float perHour(void);
    void perHour(float X);
    friend bool operator||(bool X,performance_statement &Y);
    friend bool operator&&(bool X,performance_statement &Y);
    //...
};
```

Notice that this class also provides the template class proposition with a parameter:

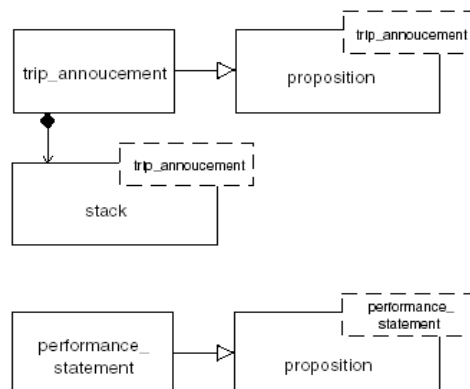
```
class performance_statement : public proposition<performance_
                                statement> {...}
```

With this declaration the proposition class is now specified for performance\_statements. The performance\_statement class is used to represent beliefs about how many sales, customers per hour, and bay utilization the owner's business has. Each statement corresponds to a single belief that the agent has in each area. This information is held in the data members:

Bays  
Sales  
PerHour

Statements such as: "Location 1 grossed \$300,000 in sales, had 10 customers per hour, and had a bay utilization of 4" can be represented by the performance\_statement class. So our agent class has two categories of beliefs implemented as datatypes derived from the proposition class. [Figure 12-2](#) shows a UML class diagram for the trip\_announcement class and the performance\_statement class. These classes hold the structure of the agent's beliefs.

**Figure 12-2. UML class diagram for the trip\_announcement class and the performance\_statement class.**



## 12.4.2 The Agent Class

The classes shown in [Figure 12-2](#) provide the foundation for the CDS of the agent, the basis for what will make the agent rational. It's the fact that the agent class is rational that distinguishes it from other types of object-oriented classes. [Example 12.4](#) shows the declaration for the agent class.

**Example 12.4 Declaration of the agent class.**

```
class agent{
    //...
private:
    performance_statement Manager1;
    performance_statement Manager2;
    performance_statement Manager3;
    trip_announcement Trip1;
    trip_announcement Trip2;
    trip_announcement Trip3;
    list<trip_announcement> TripBeliefs;
    list<performance_statement> PerformanceBeliefs;
public:
    agent(void);
    bool determineVacationAppropriate(void);
    bool scheduleVacation(void);
    void updateBeliefs(void);
    void setGoals(void);
    void displayTravelPlan(void);
    //...
};
```

As with the proposition classes, the agent class is a scaled-down version of what would be used in practice. A complete listing of the declaration of the practical versions of these classes would be three or four pages. We show enough for exposition purposes. The agent class contains two list containers:

```
list<trip_announcement> TripBeliefs;
list<performance_statement> PerformanceBeliefs;
```

The list containers are standard C++ lists. Each list is used to hold a collection of what the agent currently believes about the world. Our simple agent world is restricted to knowledge about bus trips and sales performance of his owner's business. The contents of these two containers represent the complete knowledge and belief set of the agent. If there are statements in these lists that the agent no longer believes, the agent will remove them. If the agent uncovers new statements during the course of inference, they are added to these beliefs. The agent has ongoing access to information about bus trips and the performance of the owner's business. The agent is able to update its beliefs as necessary. In addition to beliefs, the agent has objectives, which are sometimes represented as desires in the BDI (Beliefs Desires Intentions) model. The objectives support the primary directives that the agent has been given by its client. In our case the objectives will be stored in statements:

```
performance_statement Manager1;
performance_statement Manager2;
performance_statement Manager3;
trip_announcement Trip1;
trip_announcement Trip2;
trip_announcement Trip3;
```

Keep in mind that this is an oversimplification of how objectives and directives are represented within an agent class. However, there is enough here to understand how these structures are built. The three Manager statements contain the performance goals that must be met before the owner can even consider

going on vacation. The three Trip statements contain the bus trips that the owner would like to take if a vacation is earned. The beliefs together with the directives provide the basic cognitive datatypes that the agent has. The agent's inference methods combined with these cognitive datatypes form the agent's CDS (Cognitive Data Structure). The CDS forms the rational component and the distinguishing feature of an agent class. In addition to containers that hold beliefs and structures that in turn hold directives and objectives, most practical agent classes will have containers that hold the agent's intentions, commitments, or plans. The agent gives directives by his client. The agent uses its ability to inference and acts to fulfill its directives. The inferencing and actions that an agent does often result in a container that holds intentions, commitments, or plans. Our simple agent doesn't require a container to hold intentions or plans. However, it does keep track of the route that a bus trip vacation would take. This is stored in a container called Candidates. The intentions or plans would work similarly. If our agent is able to achieve its directives, it will schedule the trip and e-mail the owner the specifics. The instant our agent object is constructed, it goes to work. [Example 12.5](#) shows an excerpt from the agent's constructor.

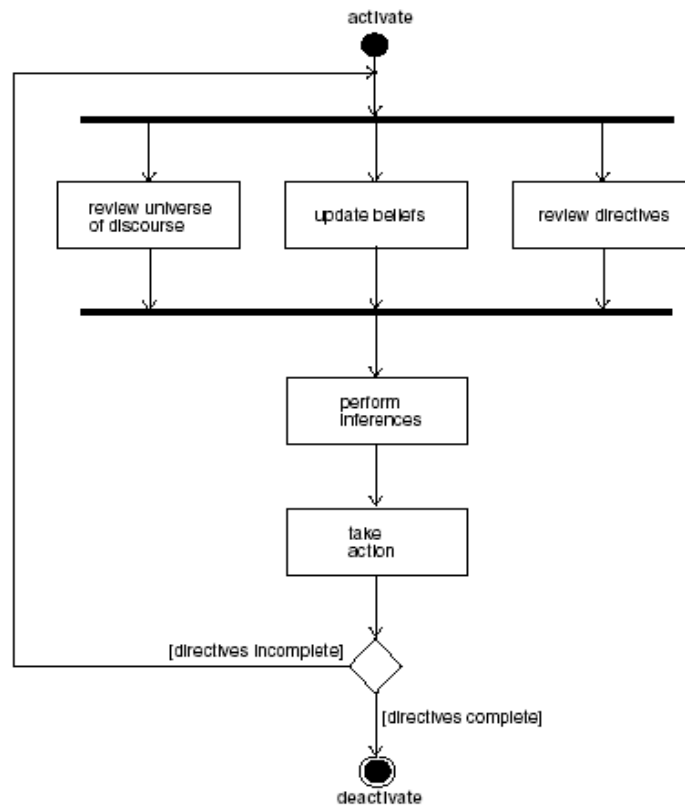
**Example 12.5 The agent class's constructor.**

```
agent::agent(void)
{
    setGoals();
    updateBeliefs();
    if(determineVacationAppropriate()){
        displayTravelPlan();
        scheduleVacation();
        cout << "Emailed Vacation Approved and Scheduled" << endl;
    }
    else{
        cout << "Emailed Vacation Not Appropriate At this time" << endl;
    }
}
```

**12.4.2.1 The Agent Loop**

Many definitions for agents include requirements of continuity and autonomy. The idea is that the agent continually performs what tasks it is assigned without the need for human intervention. The agent has the capability to interact and semi-control its environment through a feedback loop. The continuity and autonomy are often implemented as an event loop where the agent continually receives messages and events. The agent uses the messages and events to update its internal model of the world, intentions, and take action. However, autonomy and continuity are relative terms. Some agents need to function from one microsecond to the next, while other agents only need to perform their services annually. In fact, with deep space mission software, an agent may have longer than an annual cycle. Multiple years may pass before the agent performs the next task. Therefore, we don't focus on physical event loops and constantly active message queues. While these work for some agents, they are not appropriate for others. We have found the notion of a logical cycle to be the most practical. The logical cycle may or may not be implemented as an event loop. The logical cycle can be anything from nanoseconds to years. [Figure 12-3](#) shows a simple overview of a logical agent cycle.

Figure 12-3. Simple overview of a logical agent cycle.



The universe of discourse in [Figure 12-3](#) represents everything our agent can legitimately interact with. This might include files, information from ports, or data acquisition devices. The information will be represented as some kind of proposition or statement. Notice that there is a feedback loop from the agent's outputs back to the agent's inputs. Our agent from [Example 12.4](#) is only needed a few times a year. Therefore, it is not appropriate to put it in an event loop that constantly runs. Our agent will simply activate itself periodically during the course of the year to execute its initiatives. [Example 12.5](#) shows the agent's constructor. When the agent activates, it sets some goals, updates its beliefs, and then determines whether a vacation is appropriate. If the vacation is appropriate, the necessary steps are taken and the owner is e-mailed. If a vacation is not appropriate at that time, then the owner is e-mailed with that fact also.

#### 12.4.2.2 The Agent's Inference Methods

This agent has inference capabilities implemented partially by the proposition class descendants and partially by the method. Recall that the proposition class declared the operator()`=0` as a pure virtual. They force derived classes to implement the operator(). We use this operator to design a proposition so the proposition itself can determine whether it is true or not, that is, the proposition classes are self-contained. This is a fundamental tenet of object-oriented programming, namely, that a class is a self-contained encapsulation of characteristics and behaviors. Therefore, one of the primary behaviors of the proposition class and its descendants is the capability to determine whether it is true or not. The operator overloading and function objects are used to accommodate this feature. [Example 12.6](#) shows an excerpt from the proposition class and its descendant's definitions.

```
determineVacationAppropriate()
```

**Example 12.6 Excerpts from the definition of the proposition class and its descendants.**

```
template <class C> bool proposition<C>::operator&&(proposition &X)
{
    return((*this)() && X());
}

template <class C> bool proposition<C>::operator||(proposition &X)
{
    return((*this)() || X());
}
template<class C> proposition<C>::operator void*(void)
{
    return((void*)(TruthValue));
}

bool trip_announcement::operator() (void)
{
    list<trip_announcement>::iterator I;
    if(directTrip()){
        return(true);
    }
    I = UniverseOfDiscourse.begin();
    if(validTrip(I,Origin)){
        return(true);
    }
    return(false);
}
```

The definitions of the || and the && operators for the proposition classes determine whether the proposition is true or false. Each of these operator definitions ultimately calls the operator() defined for its class. Notice in [Example 12.6](#) the definition for ||. This operator is defined as:

```
template <class C> bool proposition<C>::operator||
                                   (proposition &X)
{
    return((*this)() || X());
}
```

It allows code to be written as:

```
trip_announcement A;
performance_statement B;
if (A || B){
    // Do Something
}
```

When A or B is evaluated, the operator definitions will cause the operator() to be called. Each proposition class defines behavior for the operator(). For example, the trip\_announcement class defines the operator() as:

```
bool trip_announcement::operator()(void)
{
    list<trip_announcement>::iterator I;
    if(directTrip()){
        return(true);
    }
    I = UniverseOfDiscourse.begin();
    if(validTrip(I,Origin)){
```



```

        return(true);
    }
    return(false);
}

```

This code will determine whether there is a trip from some designated origin to some destination. For example, if the desired type is Detroit to Columbus and the universe of discourse contains:

```

    Detroit to Toledo
    Toledo to Columbus

```

Then a `trip_announcement` object will report that the statement that there is a trip from Detroit to Columbus is true, although the universe of discourse does not contain a statement like:

```

    Detroit to Columbus

```

In fact, the `trip_announcement` class does check to see if there is a direct route from Detroit to Columbus. If there is a direct route, then it returns true. If there is no direct route, it attempts to find an indirect route. This behavior is accomplished by:

```

if(directTrip()){
    return(true);
}
I = UniverseOfDiscourse.begin();
if(validTrip(I,Origin)){
    return(true);
}

```

processing in the `operator()` for the `trip_announcement` class. The `directTrip()` method is straightforward and simply sequentially searches through the universe to see if there is a statement that says:

```

    Detroit to Columbus

```

The `validTrip()` method uses a DFS (Depth First Search) technique to determine if there is an indirect route. [Example 12.7](#) contains definitions for `validTrip()` and `directTrip()`:

**Example 12.7 Definitions for `validTrip` and `directTrip`.**

```

bool trip_announcement::validTrip(list<trip_announcement>::
                                iterator I, string TempOrigin)
{
    if(I == UniverseOfDiscourse.end()){
        if(Candidates.empty()){
            TruthValue = false;
            return(false);
        }
        else{
            trip_announcement Temp;
            Temp = Candidates.top();
            I = find(UniverseOfDiscourse.begin()
                    UniverseOfDiscourse.end(),Temp);
            UniverseOfDiscourse.erase(I);
            Candidates.pop();
            I = UniverseOfDiscourse.begin();
            if(I != UniverseOfDiscourse.end()){
                TempOrigin = Origin;
            }
            else{

```

```

        TruthValue = false;
        return(false);
    }

}

}
if((*I).origin() == TempOrigin && (*I).destination() == Destination){
    Candidates.push(*I);
    TruthValue = true;
    return(true);
}
if((*I).origin() == TempOrigin){
    TempOrigin = (*I).destination();
    Candidates.push(*I);
}
I++;
return(validTrip(I,TempOrigin));
}

bool trip_announcement::directTrip(void)
{
    list<trip_announcement>::iterator I;
    I = find(UniverseOfDiscourse.begin(),UniverseOfDiscourse.end(),*this);
    if(I == UniverseOfDiscourse.end()){
        TruthValue = false;
        return(false);
    }
    TruthValue = true;
    return(true);
}

```

Both the validTrip() and directTrip() methods make use of the find() algorithm from the Standard C++ library. The UniverseOfDiscourse is a container that contains the agent's beliefs and statements made to the agent. Recall that one of the first steps the agent took was to updateBeliefs(). The updateBeliefs() method is what ultimately populates the UniverseOfDiscourse container. [Example 12.8](#) contains the definition for the updateBeliefs() method.

#### Example 12.8 Update beliefs.

```

void agent::updateBeliefs(void)
{
    performance_statement TempP;
    TempP.sales(203.0);
    TempP.perHour(100.0);
    TempP.bays(4);

    PerformanceBeliefs.push_back(TempP);
    trip_announcement Temp;
    Temp.origin("Detroit");
    Temp.destination("LA");
    TripBeliefs.push_back(Temp);
    Temp.origin("LA");
    Temp.destination("NJ");
    TripBeliefs.push_back(Temp);
    Temp.origin("NJ");
    Temp.destination("Windsor");
    TripBeliefs.push_back(Temp);
}

```

In practice the beliefs will come from the agent's executing environment (i.e., files, sensors, ports, data acquisition devices, etc.). In [Example 12.8](#) the information pushed into TripBeliefs and PerformanceBeliefs represent new statements that the agent is receiving about the available trips and the performance of one of the auto repair locations. These statements will be evaluated against the directives or initiatives that the agent has. The setGoals() method establishes what the agent's directives are. [Example 12.9](#) shows the definition for the setGoals() method.

**Example 12.9 The set goals method.**

```
void agent::setGoals(void)
{
    Manager1.perHour(15.0);
    Manager1.bays(8);
    Manager1.sales(123.23);
    Manager2.perHour(25.34);
    Manager2.bays(4);
    Manager2.sales(12.33);
    Manager3.perHour(34.34);
    Manager3.sales(100000.12);
    Manager3.bays(10);
    Trip1.origin("Detroit");
    Trip1.destination("Chicago");
    Trip2.origin("Detroit");
    Trip2.destination("NY");
    Trip3.origin("Detroit");
    Trip3.destination("Windsor");
}
```

These goals tell the agent that the owner would like to go from either Detroit to Chicago, Detroit to New York, or Detroit to Windsor. In addition to the trips, the financial objectives are also set. In order for a vacation to be achieved, one or more of these objectives must be met. After the goals have been set and the agent updates its beliefs, the next objective is to determine from the goals and the beliefs if a vacation can be scheduled. The second component of the agent's inference methods is invoked:

```
determineVacationAppropriate()
```

This method will pass the UniverseOfDiscourse to each of the proposition objects. It will then use a statement of the form:

$$(A \vee B \vee C) \wedge (Q \vee R \vee S) \rightarrow W$$

which states if at least one of the statements is true from each grouping, then W is true. In the case of our agent, this means that if at least one of the sales performance goals are met and there is a bus trip that is satisfactory, then a vacation is appropriate. [Example 12.10](#) shows the definition of the determineVacationAppropriate() method.

**Example 12.10 Second inference method.**

```
bool agent::determineVacationAppropriate(void)
{
    bool TruthValue;
    Manager1.universe(PerformanceBeliefs);
    Manager2.universe(PerformanceBeliefs);
    Manager3.universe(PerformanceBeliefs);
    Trip1.universe(TripBeliefs);
    Trip2.universe(TripBeliefs);
    Trip3.universe(TripBeliefs);
    TruthValue = ((Manager1 || Manager2 || Manager3) &&
                 (Trip1 || Trip2 || Trip3));
    return(TruthValue);
}
```

Notice that the TripBeliefs and the PerformanceBeliefs are arguments to the universe() method of the Trip and Manager objects. This is where the propositions get the UniverseOfDiscourse information. Prior to the proposition calling their operator(), their UniverseOfDiscourse is populated. In [Example 12.10](#) the statement:

```
((Manager1 || Manager2 || Manager3) && (Trip1 || Trip2 ||
                                         Trip3));
```

causes six propositions to be evaluated by the || operator. The || operator for each proposition executes the operator() for each proposition. The operator() uses the UniverseOfDiscourse to determine whether the proposition is true or false. [Examples 12.6](#) and 12.7 show how the operator() is defined for the trip\_announcement class. Keep in mind that the trip\_announcement class and the performance\_statement class inherit much of their functionality from the proposition class. [Example 12.11](#) shows how the operator() is defined for the performance\_statement class.

**Example 12.11 The performance\_statement class.**

```
bool performance_statement::operator()(void)
{
    bool Satisfactory = false;
    list<performance_statement>::iterator I;
    I = UniverseOfDiscourse.begin();

    while(I != UniverseOfDiscourse.end() && !Satisfactory)
    {
        if(((I).bays() >= Bays) || ((I).sales() >= Sales) ||
           ((I).perHour() >= PerHour)){
            Satisfactory = true;
        }
        I++;
    }
    return(Satisfactory);
}
```

The operator() for each proposition class plays a part in the inferencing capabilities of the agent class. [Example 12.6](#) shows how the operator() is called whenever || or && is evaluated for a proposition class or one of its descendants. It is the combination of the proposition classes' operator() methods and the agent's methods that produce the inference methods for the agent class. In addition to the || and && operator defined by the proposition class, the trip\_announcement class and the performance\_statement

class define:

```
friend bool operator||(bool X,trip_announcement &Y);  
friend bool operator&&(bool X,trip_announcement &Y);
```

The friend declarations allow the propositions to be used in longer expressions. If we have:

```
//...  
trip_announcement A,B,C;  
bool X;  
X = A || B || C;  
//...
```

then A and B will be OR'ed together and the result will be a bool. The next part of the evaluation tries to || the bool with the trip\_announcement datatype:

```
bool || trip_announcement
```

Without the friend declararations, this would be a illegal operation. [Example 12.12](#) shows how these friend functions are defined.

**Example 12.12 Operator overloading of || and &&.**

```
bool operator||(bool X,trip_announcement &Y)  
{  
    return(X || Y());  
}  
  
bool operator&&(bool X,trip_announcement &Y)  
{  
    return(X && Y());  
}
```

Notice that the definitions of these friend functions also call the function operator() with the reference to Y(). These functions are also defined for the performance\_statement class. The goal is to make the proposition classes as easy to use as the built-in datatypes. The proposition class also defines another operator that allows the proposition to be used in a natural fashion. Let's examine the code:

```
//...  
trip_announcement A;  
if(A){  
    //... do something  
}  
//...
```

How does the compiler define Test A? The if() statement is looking for an integral type or a bool. A is neither. We want the compiler to treat A as a statement that is either true or false. The function operator is not called under this circumstance. So we define the void\* operator to give us the desired functionality. This function operator can be defined accordingly:

```
template<class C> proposition<C>::operator void*(void)  
{  
    return((void*)(TruthValue));  
}
```

This definition allows any proposition type presented in standalone fashion to be tested for a truth value. For example, when our agent class is preparing to send the owner an e-mail containing the route. The agent needs to determine which trip is available. [Example 12.13](#) shows another excerpt from the

agent's trip processing methods.

**Example 12.13 displayTravelPlan() method.**

```
void agent::displayTravelPlan(void)
{
    stack<trip_announcement> Route;
    if(Trip1){
        Route = Trip1.candidates();
    }
    if(Trip2){
        Route = Trip2.candidates();
    }
    if(Trip3){
        Route = Trip3.candidates();
    }
    while(!Route.empty())
    {
        cout << Route.top().origin() << " TO " << Route.
            top().destination() << endl;
        Route.pop();
    }
}
```

Notice that Trip1, Trip2, and Trip3 are tested as if they were bools. The candidates() method simply returns the route discovered for the trip. The operator overloading capabilities and the template facilities support the development of reusable inference methods and the CDS. These inference methods and the CDS make the object a rational object. A C++ programmer uses the classes construct to design agents. Container objects are often used in conjunction with the built-in algorithms to implement the CDS. A class that has a CDS is rational. A rational class is an agent.

### 12.4.3 Simple Autonomy

Since our simple agent class does not require the traditional "agent loop" to do its processing, we need other means to activate this agent without human intervention and on a period or cyclic basis. There will be many situations where an agent that you are writing only needs to run sometimes or only under limited conditions. The UNIX/Linux environments come with the crontab facility. The crontab facility is the user interface of the UNIX cron system. The crontab utility allows you to schedule one or more programs to be executed on a cyclic or periodic basis. Crontab jobs can be scheduled to the month, week, day, hour, and minute. To use crontab a file must be created that contains the schedule for when the agent is to be activated. The file is a simple text file set up in the following form:

minute	hour	day	month	weekday	command
--------	------	-----	-------	---------	---------

Where each column can take on the following values:

minute 0-59

hour 0-23

day 1-31

month 1-12

weekday 1-7 (1 = Mon., 7 = Sun)

command can be any UNIX/Linux command as well as the name of the file that contains your agents

Once this text file is created, it is submitted to the cron system as the command:

```
$crontab NameOfCronFile
```

For example, if we have a file named activate.agent that is set up as follows:

15	8	*	*	*	agent1
0	21	*	*	6	agent2
*	*	1	12	*	agent3

and we execute the crontab command:

```
$crontab activate.agent
```

Then agent1 will activate everyday at 8:15 A.M., agent2 will activate every Saturday at 9:00 P.M. and agent3 will activate every 1st of December. The cron files can be added or deleted as necessary. Cronfiles can contain references to other cron jobs, thus allowing an agent to reschedule itself. In fact, shell scripts can be used in conjunction with the crontab utility to provide extremely flexible, dynamic, and reliable activation of agents. See the man pages for a complete description of the crontab facility.

```
$man crontab or $man at
```

The crontab and at facilities are the simplest method to automate or regularly schedule agents that don't require constantly executing event loops. They are reliable and flexible. On the other extreme, the implementation repositories and object request brokers that we discussed in [Chapter 8](#) can also be used to implement automatic agent activation. Standard CORBA implementations also provide event looping capabilities.

## 12.5 Multiagent Systems

Multiagent systems are systems in which two or more agents cooperate, collaborate, negotiate, or compete toward the solution to some problem. The C++ software developer has several options for implementing multiagent systems. Agents can be implemented in separate threads using the POSIX thread API. This method divides a single program into multiple threads where each thread contains one or more agents. Therefore, each agent shares the same address space. This allows the agents to easily communicate using global variables and simple parameter passing. If the computer the program is running on contains more than one processor, then the agents can perform their activities in parallel. Obviously each agent should have the necessary synchronization objects defined, as discussed in [Chapters 5](#) and [11](#), and the exception handling components, as discussed in [Chapter 7](#). Multiagents implemented with multithreading are the easiest approach but limits the agents to a single computer. The most flexible approach to multiagents is using a CORBA implementation. The CORBA standard has a MAF (multi-agent facility) specification in addition to the core CORBA specification. The MICO implementation that we use in the CORBA examples in this book can be used to implement agents that can interact over the Internet, over intranets, and over local networks. The C++ binding of the CORBA standard has complete support for the object-oriented metaphor and therefore has inherent support for agent-oriented programming. In [Chapter 13](#), we discuss how the PVM and MPI libraries can be used to support agents in a parallel and distributed programming context.

### Summary

Agents are rational objects. Agent-oriented programming is another important approach to parallel and distributed programming. Agent-oriented programming provides a fresh approach to dealing with the age-old problems of decomposition, communication, and synchronization that are part of every parallel programming or distributed programming project. The C++ support for operator overloading, containers, and templates provide effective tools for implementing a wide range of agent classes. Future massively parallel and large complex distributed systems will rely on agent-oriented implementations because there is almost no other way to competently approach such systems. While the agent examples and techniques that we presented in this chapter were introductory, they provide the basis for understanding how practical agent systems can be built. The POSIX thread API, MICO, PVM, and MPI libraries that are freely available and widely used can be used to deploy multiagent systems. Multiagent systems can be used to implement either solutions that require parallel programming or solutions that require distributed programming. This book advocates two primary architectures for parallel programming and distributed programming: Agents provide the first architecture, and blackboards (which assume agents) provide the second. The next chapter provides a discussion of how to use blackboards to implement parallel and distributed programming solutions.



# Chapter 13. Blackboard Architectures Using PVM, Threads, and C++ Components

"The human brain is far more complicated than any computer, and in any event the benchmark to be attained by some super microchip of the future is to match the performance, not of an isolated human brain, but of a brain reared in a society comprising many humans..."

—Timothy Ferris, *The Universe and Eye*

In this Chapter

- [The Blackboard Model](#)
- [Approaches to Structuring the Blackboard](#)
- [The Anatomy of a Knowledge Source](#)
- [The Control Strategies for Blackboards](#)
- [Implementing the Blackboard Using CORBA Objects](#)
- [Implementing the Blackboard Using Global Objects](#)
- [Activating Knowledge Sources Using Pthreads](#)
- [Summary](#)

One of the primary goals in parallel programming is to divide the work a program must do into a set of tasks that may be executed with as much concurrency as necessary. This goal is an elusive one. Finding the correct WBS (Work Breakdown Structure) that will support parallelism and produce correct and efficient results can be a challenge. We use a modeling and architectural approach to achieve an acceptable WBS. In practice the process of modeling the problem and solution as naturally as possible with either objects or processes reveals any necessary parallelism. The model also identifies where the parallelism occurs within the problem or the solution. It's not necessary to introduce parallelism into a solution. If the problem and solution are appropriately modeled, then any necessary parallelism will be discovered. The blackboard architecture helps with this modeling process. In particular, the blackboard model helps to organize and conceptualize the concurrency and the communication within a system that requires parallelism or distributed programming.

## 13.1 The Blackboard Model

The blackboard model is an approach to collaborative problem solving. The blackboard is used to record, coordinate, and communicate the efforts of two or more software-based problem solvers. Hence there are two primary types of components in the blackboard model: the blackboard and the problem solvers.

The blackboard is a centralized object that each of the problem solvers has access to. The problem solvers may read the blackboard and change the contents of the blackboard. The contents of the blackboard at any given time will vary. The initial content of the blackboard will include the problem to be solved. Other information representing the initial state of the problem, problem constraints, goals, and objectives may be contained on the blackboard. As the problem solvers are working toward the solution, intermediate results, hypotheses, and conclusions are recorded on the blackboard. The intermediate results written by one problem solver on the blackboard may act as a catalyst for other problem solvers reading the blackboard. Tentative solutions are posted to the blackboard. If the solutions are determined not to be sufficient, these solutions are erased and other solutions are pursued. The problem solvers use the blackboard as opposed to direct communication to pass partial results and findings to each other. In some configurations the blackboard acts as a referee, informing the problem solvers when a solution has been reached or whether to start work or stop work. The blackboard is an active object, not simply a storage location. In some cases the blackboard determines which problem solvers to involve and what content to accept or reject. The blackboard may also organize the incremental or intermediate results of the problem solvers. The blackboard may translate or interpret the work from one set of problem solvers so that it may be used by another set of problem solvers.

The problem solver is a piece of software that typically has specialized knowledge or processing capabilities within some area or problem domain. The problem solver can be as simple a routine that converts from Celsius to Fahrenheit or as complex as a smart agent that handles medical diagnoses. In the blackboard model these problem solvers are called knowledge sources. To solve a problem using blackboards, two or more knowledge sources are needed and each knowledge source usually has a different area of focus or specialty. The blackboard is a natural fit for problems that can be divided into separate tasks that can be solved independently or semi-independently. In the basic blackboard configuration each problem solver tackles a different part of the problem. Each problem solver only sees the part of the problem with which it is familiar. If the solutions to any parts of the problem are dependent on the solutions or partial solutions to other parts of the problem, then the blackboard is used to coordinate the problem solvers and integrate the partial solutions. A blackboard's problem solvers need not be homogeneous. Each problem solver may be implemented using different techniques. For instance, some problem solvers might be implemented using object-oriented techniques while other solvers might be implemented as functions. Furthermore, the problem solvers may employ completely different problem-solving paradigms. For example, solver A might use a backward-chaining approach to solving its problem, while solver B might use a counterpropagation approach. There is no requirement that the blackboard's problem solvers be implemented using the same programming language.

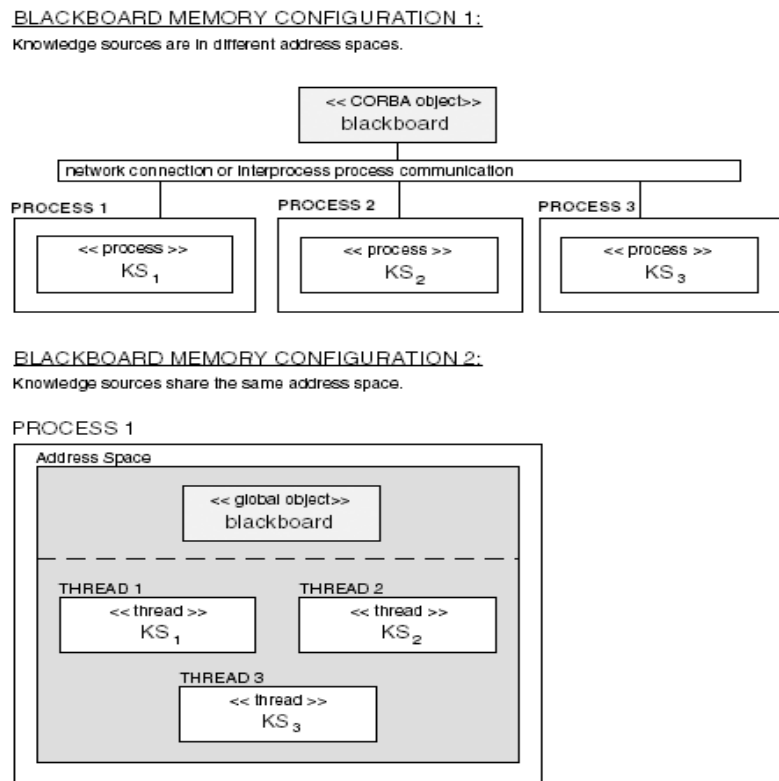
The blackboard model does not specify any particular structure or layout for the blackboard nor does it suggest how the knowledge sources should be structured. In practice, the structure of a blackboard is problem dependent.<sup>[1]</sup> The implementation of the knowledge sources is also specific to the problem being solved. The blackboard framework is a conceptual model describing relationships without describing the structures of the blackboard and knowledge sources. The blackboard model does not dictate the number or purpose of the knowledge sources. The blackboard may be a single global object or a distributed object with components on multiple computers. Blackboard systems may consist of multiple blackboards, with each blackboard dedicated to a part of the original problem. This makes the blackboard an extremely flexible model for problem solving. The blackboard model supports parallel

programming and distributed programming. First, the knowledge sources may execute simultaneously, with each knowledge source working on its part of the problem. Second, the knowledge sources may be implemented in separate threads or in separate processes on the same or different computers.

[1] While a blackboard may be reused for other, very similar problems, it is nontrivial to design a blackboard that can be used for completely different kinds or classes of problems. Reuse is usually limited to problems that are very similar in nature. This is because the solution space is closely mapped to the problem and the rule component is closely mapped to the solution space, which prevents the use of the blackboard for general problems.

The blackboard can be segmented into separate parts allowing concurrent access by multiple knowledge sources. The blackboard easily supports CREW, EREW, and MIMD. We implement the blackboard as a global object or collection of objects when the knowledge sources are implemented in separate threads. Since the threads share the same address space, a blackboard implemented as a global object or family of objects will be accessible by each threaded knowledge source. If the knowledge sources are implemented as separate processes running on the same or different computers, the blackboard is implemented as a CORBA object or collection of CORBA objects. Recall that CORBA objects can be used to support both a parallel and distributed model of computing. Here, we use CORBA to support the blackboard as a kind of distributed shared-memory between tasks executing in different address spaces. The tasks can be PVM (Parallel Virtual Machine) tasks, tasks spawned by the traditional fork-exec functions calls, or tasks spawned by the new posix\_spawn(). [Figure 13-1](#) shows our two memory configurations for the blackboard.

**Figure 13-1. Two memory configurations for the blackboard.**



In both the cases in [Figure 13-1](#), all knowledge sources have access to the blackboard. Knowledge sources in different address spaces will each make a network connection to a blackboard implemented as one or more CORBA objects. Also, when the knowledge sources are implemented as PVM tasks, the knowledge sources can supplement the blackboard communication with the message-passing model. This configuration provides for an extremely flexible model of problem solving.

## 13.2 Approaches to Structuring the Blackboard

There is no one way to structure a blackboard. However, most blackboards will have certain characteristics and attributes in common. The original contents of the blackboard will typically contain some kind of partitioning of the solution space for the problem that is to be solved. The solution space will contain all the partial solutions and full solutions to a problem. For instance, let's say that we have a search engine that searches the Internet for pictures of cars. The search engine can process a bitmap image or vector image to determine whether it contains a picture of a car and if so, whether it is the target car. Let's say that our search engine is developed using the blackboard model. Each knowledge source has a speciality: one knowledge source is a specialist in identifying images of tires, another focuses on identifying rearview mirrors, and another is an expert in identifying car door handles, lug nuts, and so on. Each aspect of the car represents a small part of the solution space. Parts of the solution space contain full images of cars from different perspectives that is, from the top, the bottom, 45-degree angles, and so on. Other parts of the solution space only contain sections of cars, perhaps the front end, the roof, the trunk, and the back end. A bitmap or vector image is placed on the blackboard and the individual knowledge sources attempt to identify something in the image that might be part of a car. If some part of the solution space matches something in the image, that piece of the image is written to another part of the blackboard as a partial solution. One knowledge source might put an identified car door handle on the blackboard. Another may put an identified car door on the blackboard. Once these two pieces of information have been put on the blackboard, another knowledge source may use this information to aid in identifying the front end of a car in the image. Once this has been identified, the image of the front end is placed on the blackboard. Each of these various ways to identify the image of a car represents part of the solution space.

The solution space is sometimes organized in a hierarchy. In our car example, complete images of cars might be at the top of the hierarchy and the next level may consist of various views of front ends and back ends, with the next level consisting of the doors, trunks, hoods, windshields, and wheels. Each level describes a smaller, perhaps less obvious image of some part of a car. The knowledge sources may work on multiple levels within the hierarchy simultaneously. The solution space may also be organized as a graph where each node represents some part of the solution and each edge represents the relationships between two partial solutions. The solution space may be represented as one or more matrices, with each element of the matrix containing a solution or partial solution. The solution space representation is an important component of the blackboard architecture. The nature of the problem will often determine how the solution space should be partitioned. In addition to a solution space component, blackboards typically have one or more rule (heuristic) components. The rule component is used to determine which knowledge sources to deploy and what solutions to accept or reject. The rule component can also be used to translate partial solutions from one level in the solution space hierarchy to another level. The rule component may also be used to prioritize the knowledge source approaches. Some knowledge sources may go down blind alleys. The blackboard deselects one set of knowledge sources in favor of another set. The blackboard may use the rule component to suggest to the knowledge sources a more appropriate potential hypothesis based on the partial hypothesis already generated. In addition to the solution space and rule component, the blackboard will often contain initial values, constraint values, and ancillary goals. In some cases the blackboard will contain one or more event queues used to capture input from either the problem space or the knowledge sources. [Figure 13-2](#) shows a logical layout for a basic blackboard architecture.

**Figure 13-2. The logical layout for a basic blackboard architecture.**

\* If the KS (knowledge source) is a process, communication can be over a network or IPC (interprocess communication). If the KS is a thread, communication may be parameter passing.

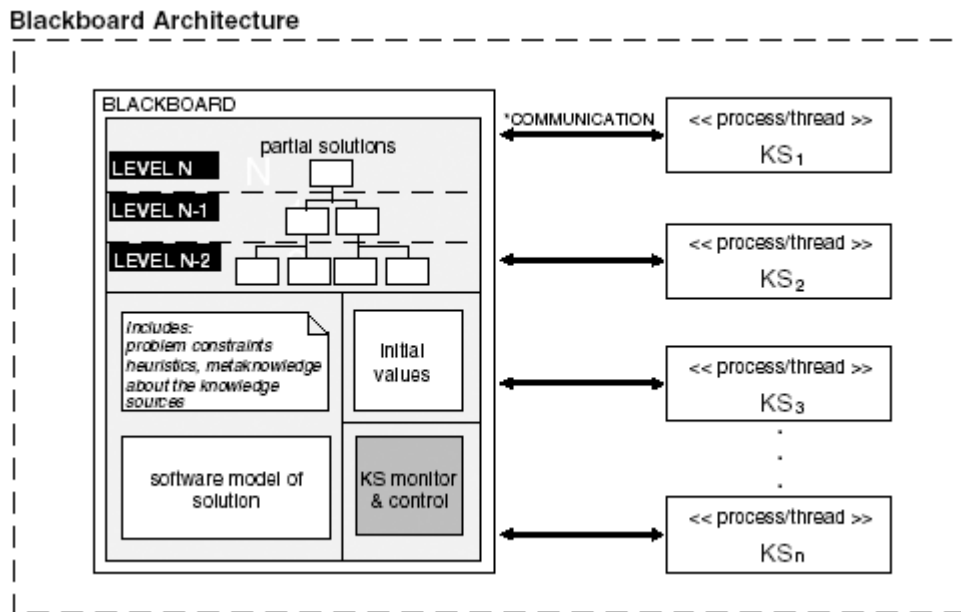


Figure 13-2 shows the blackboard has a number of segments, each segment having a variety of implementations. This suggests that blackboards are more than global pieces of memory or traditional databases. While Figure 13-2 shows the common core components that most blackboards have, the blackboard architecture is not limited to these components. Other useful components for blackboards include context models of the problem and domain models that can be used to aid the problem solvers with navigation through the solution space. The support C++ has for object-oriented design and programming fits nicely with the flexibility requirements of the blackboard model. Most blackboard architectures can be modeled using classes in C++. Recall that classes can be used to model some person, place, thing, or idea. Blackboards are used to solve problems that involve persons, places, things, or ideas. So using C++ classes to model the objects that blackboards contain or the actual blackboards is a natural fit. We take advantage of C++ container classes and the standard algorithms in our implementations of the blackboard model. In addition to the built-in classes we construct interface classes for the mutexes and other synchronization variables that we use with the blackboard. Because multiple knowledge sources can access the blackboard simultaneously, this means that the blackboard is a critical section and access needs to be synchronized. So along with the other components that a blackboard contains, we will use synchronization objects to the blackboard.

### 13.3 The Anatomy of a Knowledge Source

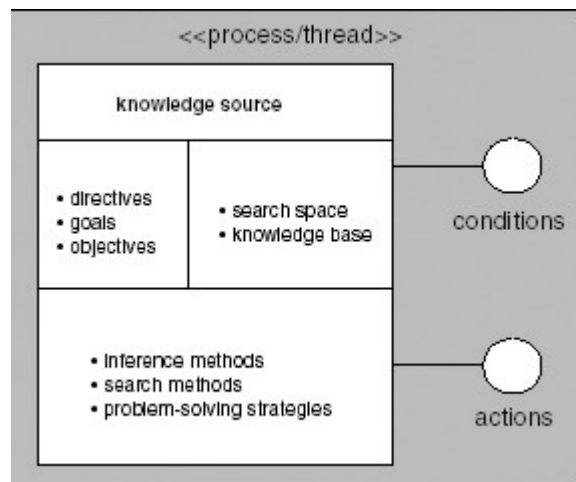
*Knowledge sources are represented as objects, procedures, sets of rules, logic assertions, and in some cases entire programs. Knowledge sources have a condition part and an action part. When the blackboard contains some information that satisfies the condition part of some knowledge source, then the action part of the knowledge source is activated. Englemore and Morgan clearly state the responsibilities of a knowledge source in their work Blackboard Systems:*

*Each knowledge source is responsible for knowing the conditions under which it can contribute to a solution. Each knowledge source has preconditions that indicate the condition on the blackboard that must exist before the body of the knowledge source is activated. One can view a knowledge source as a large rule. The major difference between*

a rule and a knowledge source is the grain size of the knowledge each holds. The condition part of this large rule is called the knowledge source precondition, and the action part is called the knowledge source body.

Here Englemore and Morgan do not specify any of the details of the condition part or the action part of a knowledge source. They are logical constructs. The condition part could be as simple as the value of some boolean flag on the blackboard or as complex as a specific sequence of events arriving in an event queue within a certain period of time. Likewise, the action part of a knowledge source can be as simple as a single statement performing an expression assignment or as involved as a forward chain in an expert system. Again, this is a statement of how flexible the blackboard model can be. The C++ class construct and the notion of an object will be sufficient for our purposes. Each knowledge source will be an object. The action part of the knowledge source will be implemented by the object's methods. The condition part of the knowledge source will be captured as data members of the object. Once the object is in a certain state then the action parts of that object will be activated. To keep things simple we will map knowledge sources to either threads or processes. Therefore, for each thread there will only be one knowledge source and for each process there will only be one knowledge source. When using the PVM with the blackboard, a knowledge source will be equivalent to a PVM task. [Figure 13-3](#) shows the logical layout of the anatomy of a knowledge source.

**Figure 13-3. The logical layout of a knowledge source.**



Each knowledge source's condition part is updated from the blackboard. Some of the knowledge source's action part updates the blackboard. Notice in [Figure 13-3](#) there is a one-for-one correlation between process space and knowledge source or thread space and knowledge source. An important attribute of the knowledge source is its autonomy. Each knowledge source is a specialist and is largely independent from the other problem solvers. This presents one of the desired qualities for a parallel program. Ideally the tasks in a parallel program can operate concurrently without much interaction with other tasks. This is exactly the case in the blackboard model. The knowledge sources act independently and any major interaction is through the blackboard. So from the knowledge source's point of view it is acting alone and getting additional information from the blackboard and recording its findings on the blackboard. The activities of the other knowledge sources and their strategies and structures are unknown. In the blackboard model, the problem is partitioned into a number of autonomous or semi-autonomous problem solvers. This is the advantage of the blackboard model over other models. In the most flexible configuration the knowledge sources are intelligent agents. The agent will be completely self-sufficient and able to act on its own with minimum interaction with the blackboard. The intelligent agent presents the greatest opportunity for large-scale parallelism.

### 13.4 The Control Strategies for Blackboards

There are several layers of control in a blackboard implementation where the knowledge sources may be activated concurrently. At the lowest layer their synchronization schemes must protect the integrity of the blackboard. The blackboard is a critical section because it is a shared, modifiable resource. In a parallel environment the knowledge source's read and write access must be coordinated and synchronized. This coordination and synchronization can involve file locking, semaphores, mutexes, and so on. This layer of control is not directly involved in the solution the knowledge sources are working toward. This is a utility layer of control and should be independent of the problem to be solved by the blackboard. In our architectural approach, this layer of control will be implemented by interface classes like the mutex, and semaphore classes that we introduced in [Chapter 11](#). Recall that the functionality contained in these classes is independent of the application they are used in. For concurrency implementations of blackboards, this layer selects one or more of the four types of parallel access that the knowledge source algorithms or heuristics will have to the physical implementation of the blackboard. That is, the users of the blackboard can be EREW, CREW, ERCW, or CRCW. This access determines how the synchronization primitives will be used. [Table 13-1](#) contains the descriptions of the four types of parallel access that a model can use.

**Table 13-1. Four Types of Parallel Access Used by a Model**

<b>PRAM models</b>	<b>Description</b>
EREW	Exclusive Read Exclusive Write
CREW	Concurrent Read Exclusive Write
ERCW	Exclusive Read Concurrent Write
CRCW	Concurrent Read Concurrent Write

The segmentation of the blackboard into parts will determine which of the types of concurrency in [Table 13-1](#) are appropriate. The most flexible CRCW can be achieved depending on the structure of the blackboard. For instance, if 16 knowledge sources are involved in a collaborative effort and each knowledge source accesses its own segment of the blackboard, then these knowledge sources can concurrently read and write the blackboard without data race problems.

The next layer of control involves the selection of which knowledge sources to involve in the search for the solution and which aspects of the problem to focus on. This is a focus-of-attention layer. This layer of control decides to focus on a certain area of the problem and selects knowledge sources accordingly. One of the major issues to tackle in any kind of problem solving is where to start and what kind of information is needed to solve the problem. The focus/attention layer evaluates the initial conditions of the problem and then controls which knowledge sources to use and where they will start. The available knowledge sources will be known to the blackboard and typically the knowledge source will accept messages or parameters that dictate how it should proceed or where in the solution space it should begin the search. For parallel implementations, this layer will determine the basic model of parallelism (distribution of the problem solvers). Usually for blackboards this is the Multiple Programs Multiple Data (MPMD, a.k.a. MIMD) model because each knowledge source/problem solver has its own area of speciality. However, the nature of the problem might warrant the popular Single Program Multiple Data

(SPMD) model. If this model is used, the control layer will spawn N number of the same knowledge source but pass different parameters to each.

The next layer of control involves determining what to do with the solution or partial solutions written to the blackboard. This layer of control will determine whether the knowledge sources can stop work or whether the solution generated is acceptable, unacceptable, partially acceptable, and so on. This layer of control has complete visibility of the blackboard and all the partial or tentative solutions. It guides the overall problem-solving strategies of the collective. As with the layout of the blackboard and the structure of the knowledge sources, the blackboard model suggests the existence of a control component but does not specify how it should be structured. Sometimes the control component is part of the blackboard. Sometimes the control component is implemented by the knowledge sources. In some cases the control component is implemented by modules external to the blackboard. The control component can also be implemented by any combination of these. The knowledge sources collectively search for a solution to some problem. We want to emphasize a solution because many problems have more than one solution. Some solutions may be deeper in the search space than others, some may cost more to find than others, and some may be deemed not good enough. The control component helps to manage the collective search strategies of the knowledge sources and monitors the tentative or partial solutions to make sure that the knowledge sources are not pursuing an impractical search strategy. The control component looks out for any infinite loops, blind alleys, or recursive regression. Furthermore, the control component is involved in selecting the best or the most appropriate knowledge sources for the problem. As the knowledge sources make progress toward a solution, the control component may relieve some knowledge sources while assigning others. The control strategy will be closely related to the search strategies used by the knowledge sources. It is important to remember that the knowledge sources may each use different search strategies and problem-solving techniques. Although they work with a common blackboard, the knowledge sources or problem solvers are essentially autonomous and self-contained. Therefore, this layer of control has a two-way communication with the knowledge sources. [Figure 13-4](#) shows possible control configurations and their layers in a blackboard architecture.

### 13.5 Implementing the Blackboard Using CORBA Objects

Recall from [Chapter 8](#) that a CORBA object is a platform-independent distributed object. CORBA objects can be accessed between processes on the same machines or processes running on different machines connected to a network. This makes CORBA objects candidates for use in PVM environments where the program is divided into a number of processes that may or may not be running on the same computer. Ordinarily the PVM environment is used for the message-passing strengths and any shared memory approaches are secondary if used at all. The notion of a network-accessible shared object adds computational power to the PVM environment. Keep in mind that CORBA objects can model anything that nondistributed objects can represent. This means PVM tasks that have shared access to CORBA objects can access container objects, framework objects, pattern objects, domain objects, and any kind of utility object. In this case, we want the PVM tasks to have access to blackboard objects. So the message-passing model is supplemented with shared access to complex objects. In addition to PVM tasks accessing distributed CORBA objects, traditional tasks spawned by the `posix_spawn()` or `fork-exec` functions can access the CORBA objects. These tasks execute in separate address spaces on the same machine but may still connect to a CORBA object that is either located on the same machine or some different machine. So while the tasks created with the `posix_spawn()` and `fork-exec` functions will all reside on the same machine, the CORBA objects can be located on any machine.



### 13.5.1 The CORBA Blackboard: An Example

To demonstrate our notion of a CORBA-based blackboard, we'll look at a blackboard developed at Ctest Laboratories. While providing a complete implementation of the blackboard is beyond the scope of this book and subject to other restrictions, we look closely at some of the most important aspects of the blackboard and the knowledge sources as they relate to our architectural approach to parallel programming. The blackboard implements a software-based course adviser. The blackboard solves the course scheduling problems for the typical college student. Students often encounter several obstacles to the perfect schedule. During course registration there is always a competition for seats in a class. At some point important classes are closed. So there is the infamous first-come, first-serve issue. So during registration where tens of thousands of students are trying to sign up for a limited number of courses, this timeliness is a factor. The student wishes to get courses that apply directly to the degree sought. Ideally these courses will be during hours that the student can attend and has open. Also, the student would like to stay within a certain course load, and keep some time open for working and other extracurricular activities. The problem is that when the student is ready to take a given course, the course may not always be offered, so substitutes or filler classes are offered to the student instead. The substitutes and filler classes add to the cost and duration of the student's education. Adding to the cost and duration are negative outcomes from the student's vantage point. However, if the substitutes or filler classes are in some way related to the student's nonacademic interests, hobbies, or goals, then the substitutes or filler classes will be reluctantly tolerated. Also, there are a number of electives and options that can be taken under the degree sought. The student wishes to have the optimum mix of courses that will allow the student to graduate either early or on time, within budget, and with the most flexibility possible. The student uses real-time course advisement software built with blackboard technology to solve the problem.

It is important to note that the blackboard has real-time access to the student's academic record, the current courses open or closed at any instant during the registration process. The blackboard has access to the student's degree plan, the university's requirements for the degree plan, the student availability schedule, the student's goals, and so on. Each of these items are modeled using C++ classes and CORBA classes and make up the components of the blackboard. To keep our blackboard example simple, we will look only at these four knowledge sources:

- General requirements counselor
- Major requirements counselor
- Electives counselor
- Minor requirements counselor

[Example 13.1](#) shows an excerpt from the blackboard's CORBA interface.

**Example 13.1 Two CORBA declarations necessary for our blackboard class.**

```
typedef sequence<long> courses;

interface black_board{
    //...
    void suggestionsForMajor(in courses Major);
    void suggestionsForMinor(in courses Minor);
    void suggestionsForGeneral(in courses General);
    void suggestionsForElectives(in courses Electives);
    courses currentDegreePlan();
    courses suggestedSchedule();
    //... };
```

The primary purpose of the `black_board` interface is to provide read/write access to the knowledge sources. In this case, the blackboard's partitions will include segments for each knowledge source.<sup>[2]</sup> This will allow the knowledge sources to access the blackboard with a CRCW policy with respect to each other. That is, multiple types of knowledge sources can access the blackboard at the same time, however, two or more knowledge sources of the same type will be restricted to a CREW policy. Any method or member function the knowledge sources will access should be defined in the `black_board` interface class. The class `courses` has been declared as a CORBA type and therefore it can be used as parameter and return values between the knowledge sources and the blackboard. So the `black_board` class declarations such as:

[2] In practice, each of the knowledge source segments will contain one or more standard C++ container classes used as data queues and event queues. Each container is made safe with synchronization components.

```
courses Minor;  
courses Major;
```

will be used to represent information that is either being written to or read from the blackboard. The type `courses` is a CORBA typedef for `sequence<long>`. A sequence in CORBA is a variable-length vector (array). This means that `courses` will be used to store an array of longs. Each long will be used to store a course code. Each course code represents a course offered at the university. Since C++ does not have a sequence type, the `sequence<long>` declaration is mapped to a C++ class. The class has the same name as `sequence<long>` typedef: `courses`. The mapping process from CORBA types to C++ types occurs during the idl compilation phase when building a CORBA application. The idl compiler will translate the `sequence<long>` declaration into C++ code. The C++ `courses` class will have a number of method functions automatically included:

```
allocbuf()  
freebuf()  
get_buffer()  
length()  
operator[]  
release()  
replace()  
maximum()
```

The knowledge sources will interact with these methods. The `sequence<long>` declaration will be transparent to the knowledge sources; they only see the class `courses`. Because CORBA supports datatypes such as structs, classes, arrays, and sequences, the knowledge sources can exchange sophisticated objects with the blackboard. This allows the programmer to maintain the object-oriented metaphor when exchanging data with the blackboard. Maintaining the object-oriented metaphor where necessary is an important part of reducing the complexity of parallel programming. The ability to easily read and write complex objects or object hierarchies from the blackboard simplifies the programming in parallel applications. There is no need to perform the translation from primitive datatypes to complex objects. The complex objects may be exchanged directly.

### 13.5.2 The Implementation of the `black_board` Interface Class

Notice in [Example 13.1](#) the interface class does not declare any variables. Recall the interface class in CORBA is restricted to only declaring the method interfaces. There are no storage components in the interface class. CORBA classes must be supplied with C++ implementations before any work can get done. The actual implementations of the methods and any variables are added to a derived class of the interface class. [Example 13.2](#) shows the derived (implementation) class for the `black_board` interface class.

**Example 13.2** An excerpt from the implementation class for the `black_board` interface class.

```
#include "black_board.h"
#include <set.h>

class blackboard : virtual public POA_black_board{
protected:
    //...
    set<long> SuggestionForMajor;
    set<long> SuggestionForMinor;
    set<long> SuggestionForGeneral;
    set<long> SuggestionForElective;
    courses Schedule;
    courses DegreePlan;
public:
    blackboard(void);
    ~blackboard(void);
    void suggestionsForMajor(const courses &X);
    void suggestionsForMinor(const courses &X);
    void suggestionsForGeneral(const courses &X);
    void suggestionsForElectives(const courses &X);
    courses *currentDegreePlan(void);
    courses *suggestedSchedule(void);
    //...
};
```

The implementation class is used to provide the actual implementations of the methods defined in the interface class. In addition to method implementation, the derived class may contain data components since it is not declared as an interface. Notice that the `black_board` implementation class in [Example 13.2](#) does not directly inherit the `black_board` interface class. Instead it inherits `POA_black_board`, which is one of the classes that the idl compiler created on behalf of the `black_board` interface class. [Example 13.3](#) contains the declaration of `POA_black_board` created by the idl compiler.

**Example 13.3** Excerpt of the `POA_black_board` class created by the idl compiler for the `black_board` interface class.

```
class POA_black_board : virtual public PortableServer:
                        :StaticImplementation
{
public:
    virtual ~POA_black_board ();
    black_board_ptr_this ();
    bool dispatch (CORBA::StaticServerRequest_ptr);
    virtual void invoke (CORBA::StaticServerRequest_ptr);
    virtual CORBA::Boolean _is_a (const char *);
    virtual CORBA::InterfaceDef_ptr _get_interface ();
    virtual CORBA::RepositoryId _primary_interface
        (const PortableServer::ObjectId &,
         PortableServer::POA_ptr);

    virtual void * _narrow_helper (const char *);
    static POA_black_board * _narrow (PortableServer::Servant);
    virtual CORBA::Object_ptr _make_stub (PortableServer::
                                         POA_ptr,
                                         CORBA::Object_ptr);

    //...
    virtual void suggestionsForMajor (const courses& Major)
        = 0;
    virtual void suggestionsForMinor (const courses& Minor)
        = 0;
```

```

virtual void suggestionsForGeneral (const courses& General)
                                = 0;
virtual void suggestionsForElectives (const courses& Electives)
                                = 0;

virtual courses* currentDegreePlan() = 0;
virtual courses* suggestedSchedule() = 0;
//...
protected:
    POA_black_board () {};
private:
    POA_black_board (const POA_black_board &);
    void operator= (const POA_black_board &);
};

```

Notice that the class in [Example 13.3](#) is an abstract virtual class because it has pure virtual member functions such as:

```
virtual courses* suggestedSchedule() = 0;
```

This means that this class cannot be used directly. It must have a derived class that provides actual member functions for every pure virtual member function. The blackboard class in [Example 13.2](#) provides the required definitions for each pure virtual member function. In the case of our blackboard class, C++ methods will be used to implement the functionality of the blackboard and invocation of the knowledge sources. However, the knowledge sources themselves are implemented partially in C++ and partially in the logic programming language Prolog.<sup>[3]</sup> But since C++ supports multilanguage and multiparadigm development, the advantages of Prolog can be intermixed with C++. In C++ we can either spawn Prolog executables using `posix_spawn()`, `fork-exec` functions, or we can access the Prolog through its foreign language interface that allows Prolog to talk directly to C++ and vice versa. Whether the actual implementation is in C++ or Prolog, the blackboard class only has to interact with C++ methods.

[3] This configuration is useful because Prolog has many features built in, such as unification, backtracking, and support for predicate logic that would have to be implemented from scratch in C++. For the examples in this book where we intermix C++ with Prolog, SWI-Prolog (University of Amsterdam) and its C++ interface library is used.

### 13.5.3 Spawning the Knowledge Sources in the Blackboard's Constructor

The blackboard is implemented as a distributed object using the CORBA protocol. One of the primary functions of the blackboard in this case is to spawn the knowledge sources. This is important because the blackboard will need access to the process ids of the tasks. The initial state of the blackboard is set in the constructor. The initial state includes information about the student, the student's academic record, the current semester, degree requirements and so on. The blackboard decides which knowledge sources to begin based on the initial state. As the blackboard evaluates the initial problem and state of the system it decides on a list of knowledge sources to invoke. Each knowledge source has an associated binary file. The blackboard uses a container called Solvers to store the names of the binaries of the knowledge sources. Later during the construction process, a function object and the `for_each()` algorithm are used to spawn the knowledge sources. Recall that any class that has the `operator()` defined can be used as a function object. Function objects are used with the standard algorithms in place of functions or in addition to functions. Usually where a function can be used a function object may be used instead. To define your own function object you must define the `operator()` with the appropriate meaning, parameter list, and return type. Our CORBA blackboard implementation can support knowledge sources implemented within PVM tasks, traditional UNIX/Linux tasks, or within separate

threads using the POSIX thread libraries. The type of task spawned in the constructor determines whether the blackboard will be working with POSIX threads, traditional UNIX/Linux processes, or PVM tasks.

### 13.5.3.1 Spawning Knowledge Sources Using PVM Tasks

Part of the constructor contains the call:

```
for_each(Solve.begin(),Solve.end(),Task);
```

The `for_each()` algorithm applies the function object operator for the task class to each element of the `Solve` container. This technique is used to spawn knowledge sources in a MIMD model, which is used when the knowledge sources each have a different speciality working on different data. [Example 13.4](#) contains the declaration of the task class.

**Example 13.4** The declaration of the task class.

```
class task{
    int Tid[4];
    int N;
    //...
public:
    //...
    task(void) { N = 0; }
    void operator()(string X);
};

void task::operator()(string X)
{
    int cc;
    pvm_mytid();
    cc = pvm_spawn(const_cast<char *>(X.data()),NULL,0,"",1,
                  &Tid[N]);N++;
}

blackboard::blackboard(void)
{
    task Task;

    vector<string> Solve;
    //...
    // Determine which KS to invoke
    //...
    Solve.push_back(KS1);
    Solve.push_back(KS2);
    Solve.push_back(KS3);
    Solve.push_back(KS4);
    for_each(Solve.begin(),Solve.end(),Task);
}
```

This `Task` class encapsulates a process that has been spawned. It will contain the task ids in the case of PVM. In the case of standard UNIX/Linux processes or Pthreads, it will contain the process ids and thread ids. This class acts as an interface to the created thread or process and the blackboard. The blackboard acts as the primary control component. It can manage the PVM tasks through their task ids. Also, the blackboard can use the PVM group operations to synchronize the PVM tasks with barriers, organize the PVM tasks into logical groups that will work on certain aspects of the problem, and to

signal group members with certain message tags. [Table 13-2](#) contains the PVM group routines and their descriptions.

The `pvm_barrier()` and the `pvm_joiningroup()` routines in [Table 13-2](#) are of particular interest to our blackboard because there are situations where the blackboard does not want to launch new knowledge sources until a certain group of knowledge sources has finished their work. The `pvm_barrier()` routine can be used to block the calling process until the appropriate knowledge sources have finished their processing. For instance, the course advisor blackboard does not want to activate the scheduler knowledge source until the knowledge sources that focus on major requirements, general requirements, minor requirements, and electives are through making their suggestions. So the blackboard will use the `pvm_barrier()` routine to wait on the PVM task group that focuses on requirements. [Figure 13-5](#) shows a UML activity diagram that shows how the knowledge sources and the blackboard are synchronized.

**Table 13-2. PVM Group Routines**

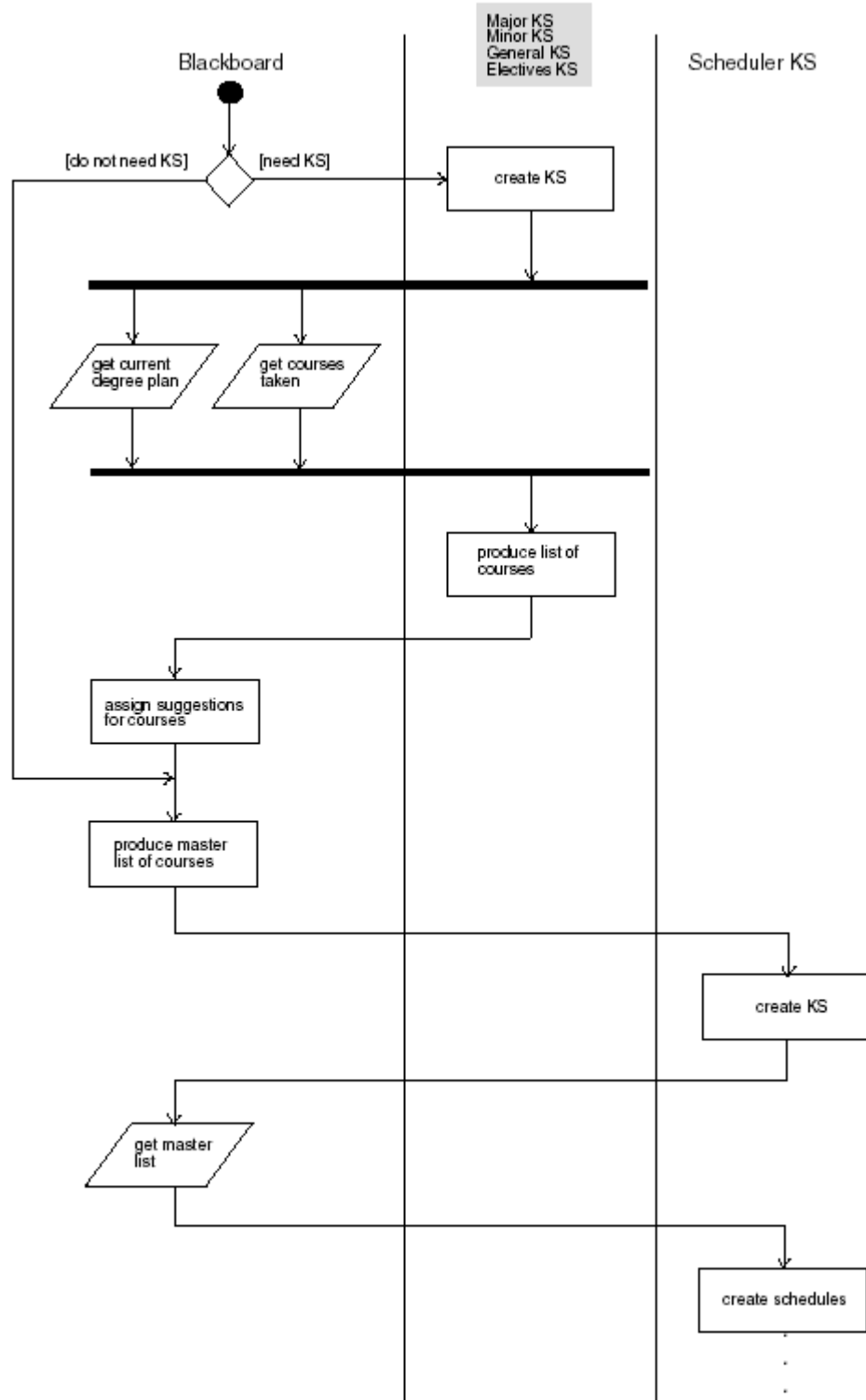
<b>PVM group operations</b>	<b>Description</b>
<code>int pvm_joiningroup(char *groupname);</code>	Enrolls the calling process in the group <code>groupname</code> and then returns an <code>int</code> , which is the instance number of this process in this group.
<code>int pvm_lvgroup(char *groupname);</code>	Unrolls the calling process from the group <code>groupname</code> .
<code>int pvm_gsize(char *groupname);</code>	Returns an <code>int</code> , which is the number of members in the group <code>groupname</code> .
<code>int pvm_gettid(char *groupname, int inum);</code>	Returns an <code>int</code> , which is the task id of the process identified by the group name <code>groupname</code> and the instance number <code>inum</code> .
<code>int pvm_getinst(char *groupname, int taskid);</code>	Returns an <code>int</code> , which is the instance number associated with the group name <code>group-name</code> and the process with the task id <code>taskid</code> .
<code>int pvm_barrier(char *groupname, int count);</code>	Blocks the calling process until <code>count</code> members in the group <code>groupname</code> have called this function.
<code>int pvm_bcast(char *groupname, int messageid);</code>	Broadcasts a message stored in the active send buffer associated with <code>messageid</code> to all the members of the group <code>groupname</code> .
<code>int pvm_reduce(void *operation, void *buffer, int count, int datatype, int messageid, char *groupname,</code>	Performs a global operation <code>operation</code> on all the processes in the group <code>groupname</code> .

# PVM group operations

# Description

```
int root);
```

Figure 13-5. UML activity diagram showing the synchronization of the blackboard and the knowledge sources.



In [Figure 13-5](#), the synchronization barrier is implemented with the help of the `pvm_barrier()` and

pvm\_joiningroup() routines. [Example 13.5](#) contains the function operator of the task object.

**Example 13.5 The function operator of the task object.**

```
void task::operator()(string X)
{
    int cc;
    pvm_mytid();
    cc = pvm_spawn(const_cast<char *>(X.data()),NULL,0,"",1,
                  &Tid[N]);N++;
}
```

The function operator is used to spawn PVM tasks. The name of the task is contained in X.data(). The call to the pvm\_spawn() routine in [Example 13.5](#) creates one task and stores the task id of the task in Tid[N]. The pvm\_spawn() routine and the invocation of PVM tasks are discussed in [Chapter 6](#). The task class is used as a function object. The algorithm:

```
for_each(Solve.begin(),Solve.end(),Task);
```

will cause the operator() to execute for the Task object. This operation will cause a knowledge source in the Solve container to be activated. The for\_each() algorithm ensures that each knowledge source will be activated. If the SIMD model is used, then the for\_each() algorithm is not necessary. Instead we use a PVM spawn call directly in the constructor of the blackboard. [Example 13.6](#) shows how a set of PVM tasks using a SIMD model can be launched from the blackboard constructor.

**Example 13.6 Launching PVM tasks from the task class constructor.**

```
void task::operator()(string X)
{
    int cc;
    pvm_mytid();
    cc = pvm_spawn(const_cast<char *>(X.data()),NULL,0,"",1,
                  &Tid[N]);N++;
}
```

### 13.5.3.2 Connecting Blackboard and the Knowledge Sources

In [Example 13.6](#), 20 knowledge sources are spawned. Initially they will each execute the same code. After they are spawned, the blackboard will send messages to them representing what part they are to play in the problem-solving process. With this configuration the knowledge sources and the blackboard are part of the PVM. After the knowledge sources are created, they will be able to interact with the blackboard by connecting to the port that the blackboard is located on, or to the address the blackboard is at on an intranet or the Internet. The knowledge sources will need the object reference for the blackboard. These references may be coded within the knowledge sources, or the knowledge sources might read them from a configuration file, or get them from a naming service. Once the knowledge source has the reference the knowledge source interacts with the ORB (Object Request Broker) to locate the actual knowledge and activate it. For our example, we will assign the blackboard a specific port. We start the CORBA blackboard with a command:

```
blackboard -ORBIIOPAddr inet:porthos:12458
```

This command executes our blackboard program and assigns it to listen on port 12458 on host porthos. Starting a CORBA object will differ depending on the CORBA implementation used. Here we are using Mico,[\[4\]](#) an open-source implementation of CORBA. When the blackboard program executes, it



instantiates the blackboard that in turn spawns the knowledge sources. When the knowledge sources are spawned by the blackboard they will have the port number hard-coded. [Example 13.7](#) shows an excerpt from a knowledge source that connects to the CORBA-based blackboard.

[4] We used Mico 2.3.3 in the Linux environment and Mico 2.3.7 under Solaris 8 for all of the CORBA examples in this book.

**Example 13.7 A knowledge source that connects to the CORBA blackboard.**

```
1 #include "pvm3.h"
2 using namespace std;
3 #include <iostream>
4 #include <fstream>
5 #include <string.h>
6 #include <sstream>
7 #include "black_board_impl.h"
8
9 int main(int argc, char *argv[])
10 {
11     CORBA::ORB_var Orb = CORBA::ORB_init(argc,argv,
12                                     "mico-local-orb");
13     CORBA::Object_var Obj =Orb->bind("IDL:black_board:1.0",
14                                     "inet:porthos:12458");
15     courses Courses;
16     //...
17     //...
18     black_board_var BlackBoard = black_board::_narrow(Obj);
19
20     int Pid;
21     //...
22     //...
23     cout << "created the knowledge source" << endl;
24     Courses.length(2);
25     Courses[0] = 255551;
26     Courses[1] = 253212;
27     string FileName;
28     stringstream Buffer;
29     Pid = pvm_mytid();
30     Buffer << "Result." << Pid << ends;
31     Buffer >> FileName;
32     ofstream Fout(FileName.data());
33     BlackBoard->suggestionsForMajor(Courses);
34     Fout.close();
35     pvm_exit();
36     return(0);
37 }
```

In line 11 in [Example 13.7](#), the ORB runtime is initialized. Line 12 associates the black\_board object name with the port 12458 and returns a reference to the CORBA object in the Obj variable. Line 16 performs a kind of cast operation so that the Blackboard variable is referring to the right size object. Once the knowledge source has instantiated the Blackboard object, any method declared in the black\_board interface shown in [Example 13.1](#) may be invoked. Notice on line 13 that the object Courses is instantiated. Recall that courses was originally defined as a CORBA sequence type. Here, the knowledge source is using the class courses created during the idl compilation. The elements are added to this class as they would be for an array.

Lines 24 and 25 add two courses to the Courses object and line 32 contains the method invocation:  
 BlackBoard->suggestionsForMajor(Courses)

This call writes the courses on the blackboard. Similarly, the:

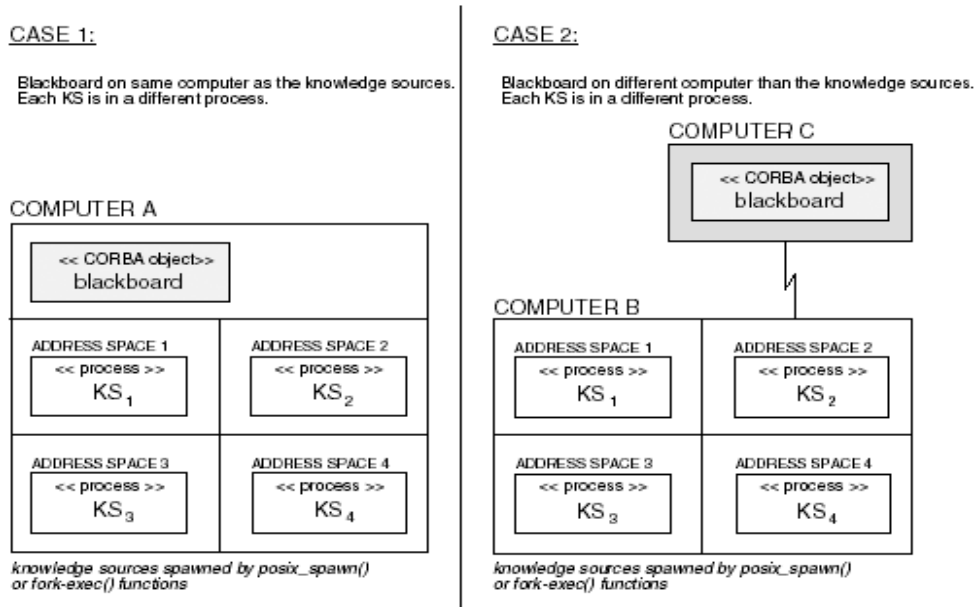
```
courses currentDegreePlan();
courses suggestedSchedule();
```

methods can be used to read information from the blackboard. So all that is needed by the knowledge source is a reference to the Black\_board object. The Black\_board object may be located anywhere within an intranet or the Internet. It is the ORB's responsibility to actually locate the object. (Chapter 8 discusses the process of locating and activating CORBA objects.) Because the Black\_board object has the PVM task ids, it may perform task management and send and receive messages directly from the knowledge sources. Likewise the knowledge sources may communicate directly with each other using the more traditional PVM messaging. It is important to note that the destructor for the Black\_board object will call pvm\_exit() and each knowledge source should call pvm\_exit() after there are no more PVM system calls. This will remove them from the PVM environment, although other processing may continue.

### 13.5.3.3 Activating Knowledge Sources Using POSIX spawn()

Implementing the knowledge sources or problem solvers within PVM tasks is especially useful when the tasks will run on separate computers. Each knowledge source can take advantage of any special resource that a particular computer may have. These resources can include processor speed, databases, special peripherals, processor architectures, and numbers of processors. The PVM tasks may also be used on a single computer that has multiple processors. However, since our blackboard can be implemented by connecting to ports, we can just as easily use traditional UNIX/Linux processes to contain our knowledge sources. When the knowledge sources are created in standard UNIX/Linux processes and the computer has multiple processors, then the knowledge sources may run concurrently on the processors available. Obviously if there are more knowledge sources than there are processors, then multitasking will be necessary. Figure 13-6 shows two simple architectures that can be used with the CORBA-based blackboard and UNIX/Linux processes.

Figure 13-6. Two architectures that can be used with the CORBA-based blackboard and UNIX/Linux processes.



In Case 1, the CORBA object is located on the same computer as the knowledge sources and each knowledge source has its own address space. That is, each knowledge source has been spawned using either the `posix_spawn()` routine or the `fork-exec` routines. In Case 2, the CORBA object is located on a different computer and the knowledge sources are each located on the same computer but in different address spaces. In both Case 1 and Case 2, the CORBA object acts as a kind of shared memory for the knowledge sources because they each have access to it and they may exchange information through the blackboard. But there is a major advantage of the CORBA object—it is far more sophisticated than a simple block of memory locations. The blackboard is a complete object that may consist of any type of data structure, object, or even other blackboards. This kind of sophistication cannot be easily implemented using the basic shared memory routines. So the CORBA gives us an ideal method for implemented complex shared objects between processes. In [section 13.5.3.1](#) we spawned PVM tasks that implemented the knowledge sources. Here we change the constructor to contain calls to the `posix_spawn()` routine or we can use our `for_each()` algorithm and the task function object to call the `posix_spawn()` routine. In Case 1 in [Figure 13-6](#), the blackboard can spawn the knowledge sources from its constructor. However, in Case 2, this is not possible because the blackboard is located on a separate computer. In Case 2, the blackboard will use some intermediary to cause the `posix_spawn()` routine to be executed. There are several options available such as the blackboard calling another CORBA object on the computer that contains the knowledge sources, or RPC, or using a MPI or PVM task that will call a program containing a call to `posix_spawn()`. [Chapter 3](#) contains a discussion of how to set up a call to `posix_spawn()`. [Example 13.8](#) shows how the `posix_spawn()` routine can be used to activate one of the knowledge sources.

**Example 13.8 Using `posix_spawn()` to launch knowledge sources.**

```
#include <spawn.h>

blackboard::blackboard(void)
{
    //...
    pid_t Pid;
    posix_spawnattr_t M;
    posix_spawn_file_actions_t N;
    posix_spawn_attr_init(&M);
    posix_spawn_file_actions_init(&N);
    char *const argv[] = {"knowledge_source1",NULL};
    posix_spawn(&Pid,"knowledge_source1",&N,&M,argv,NULL);
    //...
}
```

In [Example 13.8](#), the spawn attributes and the spawn file actions are initialized and then the `posix_spawn()` routine is used to create a separate process that will execute `knowledge_source1`. Once this process is created the blackboard has some access to the process through the process's id stored in `Pid`. In addition to the blackboard as a means of communication, the standard IPC communication is available if the blackboard is located on the same computer as the knowledge sources. While sockets are available in the configuration where the blackboard is on a separate computer, the blackboard is the simplest means of communication between the knowledge sources. When using this configuration, the control that the blackboard has over the knowledge sources will be governed more strictly by the content of the blackboard at any given time as opposed to sending messages directly to the knowledge sources. Sending messages directly is more easily accommodated using the blackboard in conjunction with PVM tasks. Here the knowledge sources regulate themselves based on the content of the blackboard. However, the blackboard does have some level of control over the knowledge sources because the blackboard has the process ids for each process containing a knowledge source. Both the MPMD (MIMD) and SPMD (SIMD) models are also supported using the `posix_spawn()` routine.

[Example 13.9](#) shows a class that will be used as a function object with the `for_each()` algorithm.

**Example 13.9** The `child_process` will be used as a function object when launching knowledge sources.

```
class child_process{
    string Command;
    posix_spawnattr_t M;
    posix_spawn_file_actions_t N;
    pid_t Pid;
    //...
public:
    child_process(void);
    void operator()(string X);
    void spawn(string X);
};

void child_process::operator()(string X)
{
    //...
    posix_spawnattr_init(&M);
    posix_spawn_file_actions_init(&N);
    Command.append("/tmp/");
    Command.append(X);
    char *const argv[] = {const_cast<char*>(Command.data())
                          ,NULL};
    posix_spawn(&Pid,Command.data(),&N,&M,argv,NULL);
    Command.erase(Command.begin(),Command.end());
    //...
}
```

We encapsulate the attributes for the `posix_spawn()` routine in a class named `child_process`. Encapsulating all of the information needed to use the `posix_spawn()` routine within a class makes it easier to use `posix_spawn()` and provides a natural interface to the attributes of a process that will be created using `posix_spawn()`. Notice in [Example 13.9](#) that we defined the operator `()` for the `child_process` class. This means that the class can be used as a function object with the `for_each()` algorithm. As the blackboard decides which knowledge sources will be involved in a problem-solving effort, it stores the name of the knowledge sources in a container we call `Solve`. Later, during the construction of the blackboard, the knowledge sources are activated using the `for_each()` algorithm.

```
// Constructor
//...
child_process Task;
for_each(Solve.begin(),Solve.end(),Task);
```

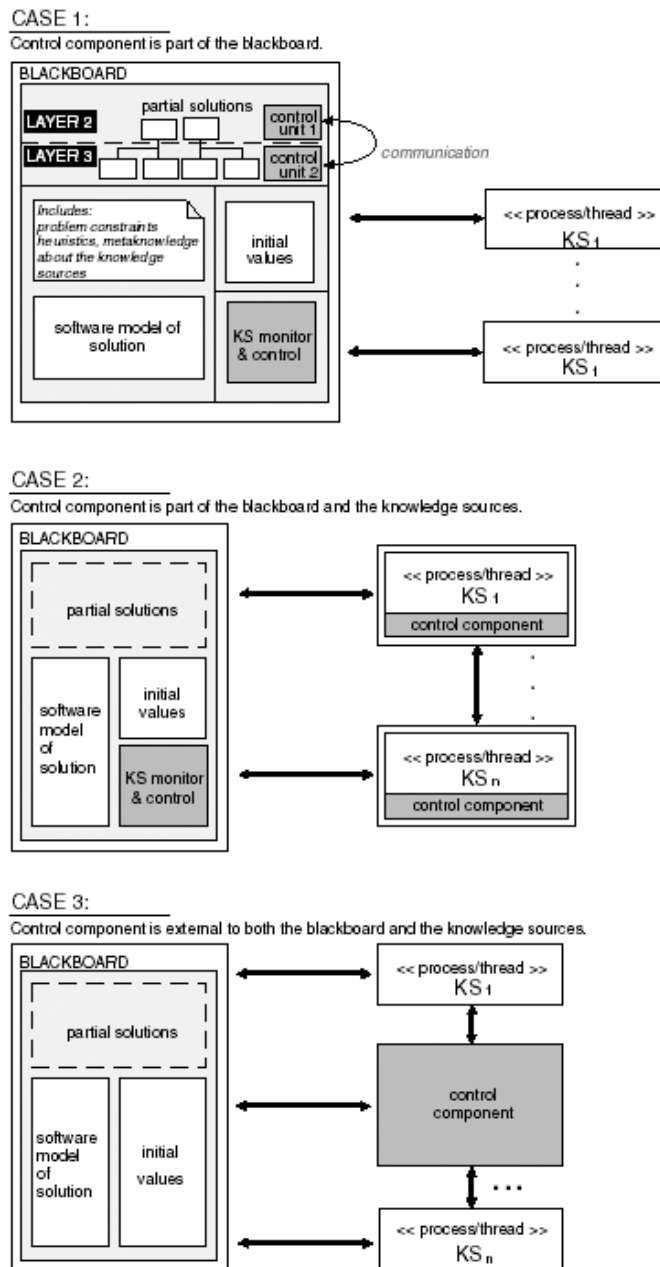
This will cause the `operator()` method shown in [Example 13.9](#) to be executed for each element of the `Solve` container. Once these knowledge sources are activated, they access a reference to the blackboard and can begin the problem-solving process. Although these are not PVM tasks they connect to the blackboard in the same manner (see [section 13.5.3.2](#)) and they perform the work in the same manner. The difference is the interprocess communication between standard UNIX/Linux processes and the interprocess communication that is possible using the PVM environment. Also, the PVM tasks may be located on separate computers. While processes created with `posix_spawn()` exist on the same computer. If processes created by either `posix_spawn()` or the `fork-exec` routines are to be used in conjunction with the SIMD model, then the `argc` and `argv` parameters can be used in addition to the blackboard to assign the knowledge sources a specific area of the problem to solve. In the case where the blackboard is on the same computer as the knowledge sources and the blackboard activates the

knowledge sources in its constructor, then technically the blackboard is the parent of the knowledge sources and the knowledge sources will inherit the environment variables of the blackboard. The environment variables of the blackboard are an additional method that can be used to pass information to the knowledge sources. These environment variables can be easily manipulated using the:

```
#include <stdlib.h>
//...
setenv();
unsetenv();
putenv();
```

routines. When the knowledge sources are implemented in processes that were created using either `posix_spawn()` or the `fork-exec` routines, the programs look like regular CORBA programs and can take advantage of all of the facilities that the CORBA protocol has to offer.

**Figure 13-4. Control configurations and their layers in the blackboard architecture.**



Note the configuration in [Figure 13-4](#) contains the control within the blackboard as opposed to a separate module or knowledge source. In this configuration, the control is designed as part of the blackboard class. Since a two-way communication is needed in Layer 2 and Layer 3, it is convenient to have the blackboard spawn the processes or threads that will contain the knowledge sources. If the blackboard spawns the processes or threads it will have easy access to the thread ids or process ids. This will allow the blackboard to easily broadcast messages to the knowledge sources, and perform process and thread management. When the blackboard needs to terminate a knowledge source for some reason, access to the thread id or process id will make this easy. Notice in [Figure 13-4](#) that one of the options is to have the control external to the blackboard and the knowledge sources. If this configuration is used, thread ids and process ids must be communicated explicitly to the control modules.

### ***13.6 Implementing the Blackboard Using Global Objects***

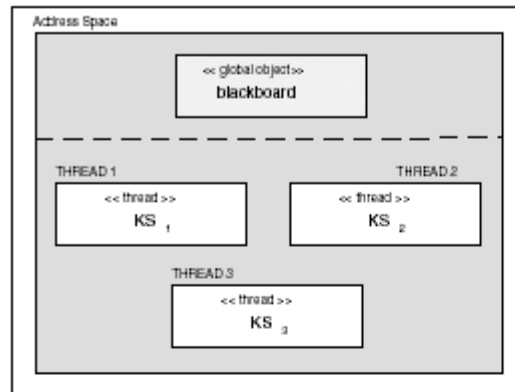
The choice of a CORBA-based blackboard is a natural choice when the knowledge sources will be implemented within an intranet or the Internet environment or when the knowledge sources will be implemented in separate processes for purposes of modularity, encapsulation, and so on. However, distributed blackboards are not always necessary. In the case where the knowledge sources can be implemented within the same process and on the same computer, multiple threads provide a superior solution because they will be faster, have less overhead, and are easier to use and set up. The communication between multiple threads is also easier because the threads share the same address space and can use global variables. In fact, with threads the blackboard will be instantiated as a global object available to all of the threads within the process. There is no need for inter-process communication, socket communication, or any other kind of network communication when the knowledge sources are implemented as threads within a single program. Also, the added layer of the CORBA protocol is not necessary and the objects may be designed as regular C++ classes. If the program is running on a machine that has multiple processors, then the threads may run concurrently on as many processors as are available. The thread's configuration of the blackboard is very attractive in SMP and MPP systems. In general, threads will have the best performance. Threads are often referred to as lightweight processes because they don't require the same overhead as traditional UNIX/Linux processes. The POSIX threads (Pthreads) library offers virtually everything needed for knowledge source creation and management. [Figure 13-7](#) contrasts the three basic configurations for process distributions for blackboards and knowledge sources.

**Figure 13-7. Contrasts of the three basic configurations for process distribution for blackboards and knowledge sources.**

**PROCESS DISTRIBUTION 1:**

Knowledge sources are implemented as threads within the same process sharing the blackboard as a global object.

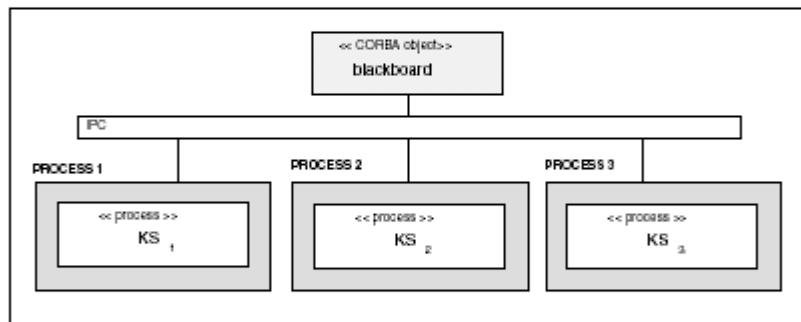
**SINGLE PROCESS**



**PROCESS DISTRIBUTION 2:**

Knowledge sources are implemented as processes running on the same computer using IPC to communicate with the Blackboard as a CORBA object running on the same computer.

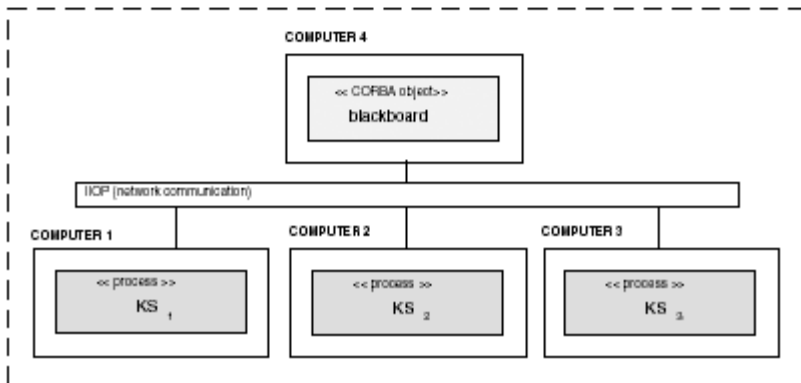
**SINGLE COMPUTER WITH MULTIPLE PROCESSES**



**PROCESS DISTRIBUTION 3:**

Knowledge sources are implemented as processes running on different computers using IIOP to communicate with the Blackboard as a CORBA object.

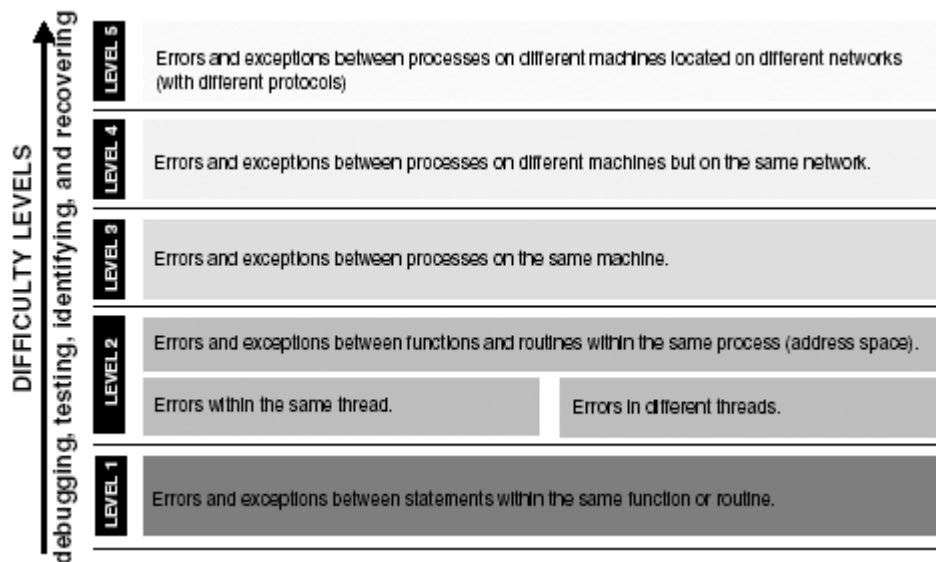
**MULTIPLE PROCESSES ON MULTIPLE COMPUTERS**



Because the blackboard is implemented within a multithread environment, then Pthread mutexes and condition variables may be used to synchronize access to the blackboard. Of course the mutexes and condition variables should be encapsulated within interface classes, as discussed in [Chapter 11](#). Also, `pthread_cond_signal()` and `pthread_cond_broadcast()` can be used to coordinate and synchronize the work that the knowledge sources are performing. Since the blackboard creates the threads it will have

easy access to the thread id of each knowledge source. This means the blackboard can cancel a thread if necessary with `pthread_cancel()`. Also, the blackboard will be able to synchronize on the various knowledge sources using the `pthread_join()` routine. In addition to the performance and ease-of-use advantages of threads and global blackboards, there is also the issue of error and exception management. In general, it is easier to deal with errors and exceptions within the same process than between different processes, and with errors on the same machine than between different machines. [Figure 13-8](#) shows the exception and error level difficulty when handling errors and difficulties within a program.

**Figure 13-8. The exception and error level difficulty.**



Since the knowledge sources are implemented within separate threads within the same process, any errors or exceptions that occur will be at Level 2. Whenever programs that require concurrency are designed and developed, the complexity of handling and recovering from errors and exceptions must be considered. The blackboard implemented as a global object and the knowledge sources implemented as threads are the simplest architecture when using the blackboard model for concurrency. [Example 13.10](#) contains an excerpt declaration from our course advisor blackboard.

**Example 13.10 An excerpt from the course advisor blackboard designed for a threaded environment.**

```
class blackboard{
protected:
    //...
    set<long> SuggestionForMajor;
    set<long> SuggestionForMinor;
    set<long> SuggestionForGeneral;
    set<long> SuggestionForElective;
    set<long> Schedule;
    set<long> DegreePlan;
    mutex Mutex[10];
    //...
```



```

public:
    blackboard(void);
    ~blackboard(void);
    void suggestionsForMajor(set<long> &X);
    void suggestionsForMinor(set<long> &X);
    void suggestionsForGeneral(set<long> &X);
    void suggestionsForElectives(set<long> &X);
    set<long> currentDegreePlan(void);
    set<long> suggestedSchedule(void);
    //...
};

```

This blackboard class is designed to be instantiated as a global object accessible to all the threads within a program. Notice that the blackboard class in [Example 13.10](#) has an array of mutexes. These mutexes will be used to protect the critical sections within the blackboard. The knowledge sources are virtually unaware of the synchronized access to the critical sections because the synchronization is encapsulated within the blackboard.

### 13.7 Activating Knowledge Sources Using Pthreads

The knowledge sources are implemented within separate threads. The blackboard's constructor creates the threads and assigns each thread a specific knowledge source. This gives the blackboard its MIMD model. [Example 13.11](#) contains part of the constructor for the blackboard.

**Example 13.11** The blackboard's constructor used to create the threads that will contain the knowledge sources.

```

blackboard::blackboard(void)
{
    pthread_t Tid[4];
    //...
    try{
        pthread_create(&Tid[0],NULL,suggestionForMajor,
                      NULL);
        pthread_create(&Tid[1],NULL,suggestionForMinor,
                      NULL);
        pthread_create(&Tid[2],NULL,suggestionForGeneral,
                      NULL);
        pthread_create(&Tid[3],NULL,suggestionForElective,
                      NULL);
        pthread_join(Tid[0],NULL);
        pthread_join(Tid[1],NULL);
        pthread_join(Tid[2],NULL);
        pthread_join(Tid[3],NULL);
    }
    //...
}

```

Notice that the constructor calls the `pthread_join()` routine. This causes the constructor to wait for these four threads to terminate before it continues. These threads can be activated from other member functions of the blackboard. However, the particular processing these knowledge sources are performing for the constructor is a preliminary kind of initialization for the blackboard, so it is totally appropriate for the blackboard to wait on these threads before it continues to construct the object. This technique of creating threads within the constructor also raises error handling and exception handling issues. What happens if the threads fail for some reason? Since constructors don't have return values, exception handling must be used.

Each thread is associated with a function. In this case:

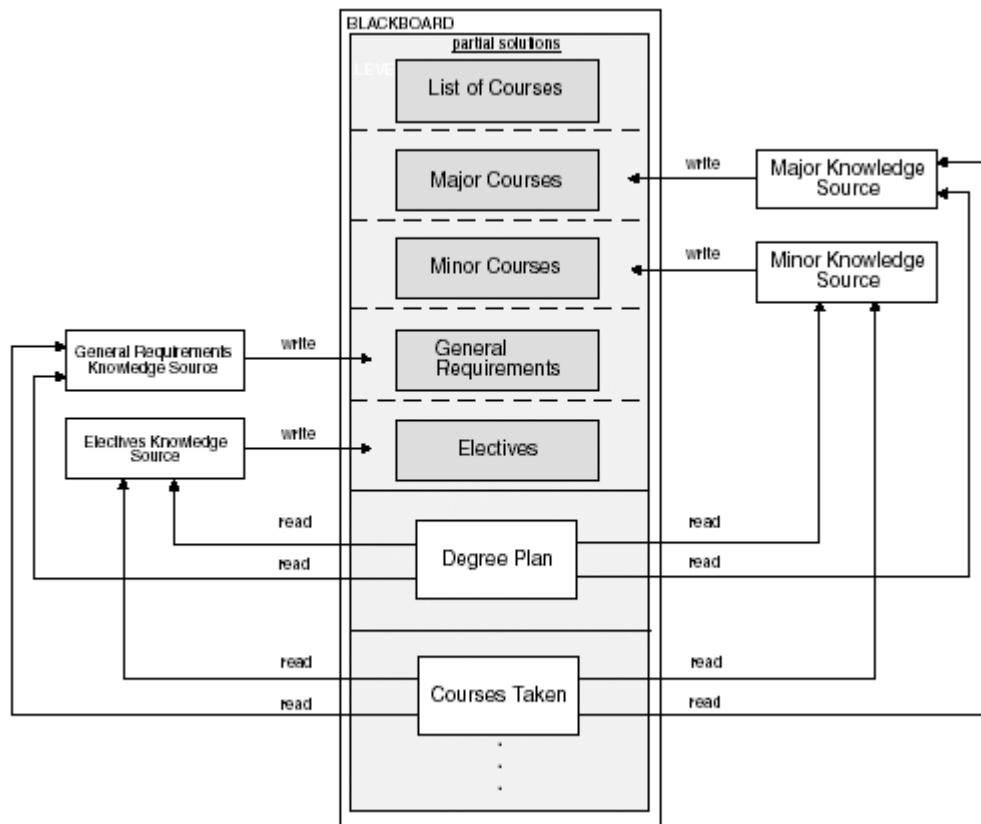
```
void *suggestionForMajor(void *X);  
void *suggestionForMinor(void *X);  
void *suggestionForGeneral(void *X);  
void *suggestionForElective(void *X);
```

These four functions are used by the threads to implement the functionality of these particular knowledge sources. Since the blackboard is a global object, each of these functions has immediate access to the blackboard's member functions. So the knowledge sources may call the blackboard's member functions directly, such as:

```
//...  
Combination.generateCombinations(1,9,Courses);  
Result = Combination.element(9);  
//...  
Blackboard.suggestionsForMinor(Value);  
//...
```

Since some divisions of the blackboard are restricted to a particular knowledge source, these divisions of the blackboard may be accessed using CRCW policies, as shown in [Figure 13-9](#).

**Figure 13-9. The four knowledge sources may concurrently read and write their section of the blackboard.**



The type of parallelism shown in [Figure 13-9](#) is a natural scheme in blackboard systems because the blackboard is often divided into sections, with each section referring to a certain part of the problem or subproblem. There is typically one knowledge source per problem area, so these sections may be accessed concurrently.

## ***Summary***

The blackboard model supports concurrency. The concurrency is inherent in the structure of the blackboard and the relationship between the blackboard and the knowledge sources and between the knowledge sources and each other. The blackboard is a problem-solving model. The problem is divided up into knowledge-specific areas. Each area is assigned a knowledge source or problem solver. The knowledge sources and problem solvers are typically self-contained and require little interaction with the other knowledge sources. The communication that is necessary occurs through the blackboard. Therefore, the knowledge sources and problem solvers serve to modularize the processing within the program. These modules can be treated separately and they can execute concurrently without complex synchronization needs. The blackboard may be implemented using CORBA objects. When the blackboard is implemented as CORBA objects, the knowledge sources can be distributed across intranets or the Internet. The blackboard acts as a kind of shared distributed memory for tasks within a PVM-type environment. The blackboard model easily supports MPMD (MIMD) and the SPMD (SIMD) model. The concept of the blackboard motivates the designer to break down the work that a program needs to do along knowledge lines. This results in the program having a WBS of knowledge specialists. The blackboard will contain software models of the problem domain and the solution space. These software models help the designer and developer to discover any parallelism that is necessary within a program that will be implemented. Alongside of the classic client-server model of distributed programming the blackboard model is one of the most powerful models available for both distributed and parallel programming. The knowledge sources or problem solvers in the blackboard model are often implemented as agents. In the next chapter we will take a closer look at how to implement agents and how to deploy multiagent systems to achieve concurrency.

## ***Appendix A. Diagrams***

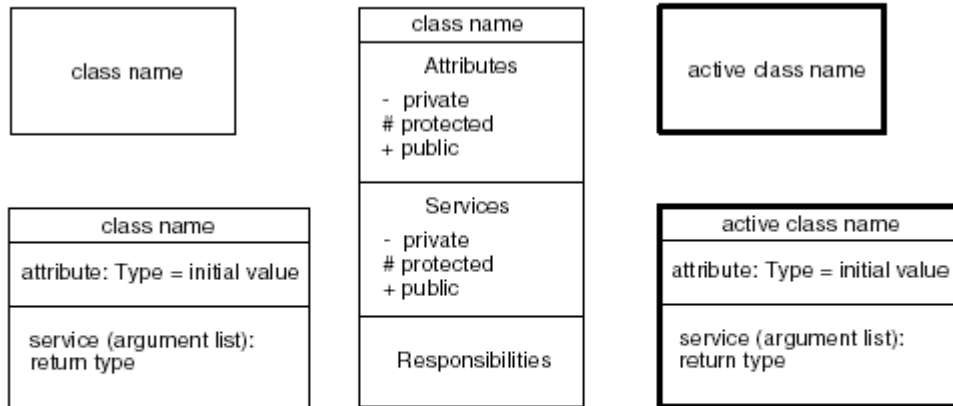
This appendix provides a quick reference to the UML diagrams used throughout this book. The UML (United Modeling Language) is a graphical notation used to design, visualize, model, and document the artifacts of a software system. It is the de facto standard for communicating and modeling object-oriented systems. The modeling language uses symbols and notations to represent the artifacts of a software system from different views and different focuses. Although there are other graphical notations and artifacts used in this book, this appendix provides a quick way the reader can become familiar with the basic UML notations and symbols they may require in documenting their software systems.

### ***A.1 Class and Object Diagrams***

Class and object diagrams are the most common diagrams used in modeling an object-oriented system. Class diagrams can be used to represent each type of class in your system, including template classes and interface classes. Class diagrams can include the details of the class (e.g., attributes, services). Class and object diagrams can show the datatype, value of variables, and return types of functions. Object diagrams can show the object name. Both types of diagrams can depict the number of classes or objects used in the system along with their relationships between classes and objects.

Figure A-1. The multiple ways a class or object can be represented. Classes can show services, attributes, and visibility. Active classes or objects use a heavier line.

REPRESENTING A CLASS



REPRESENTING AN OBJECT

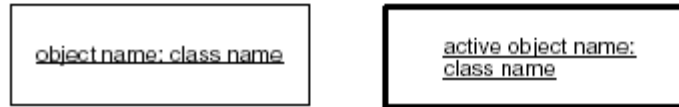


Figure A-2. Multiple instances of classes and objects. Multiple instances can be shown graphically or by using multiplicity notation.

REPRESENTING MULTIPLE INSTANCES

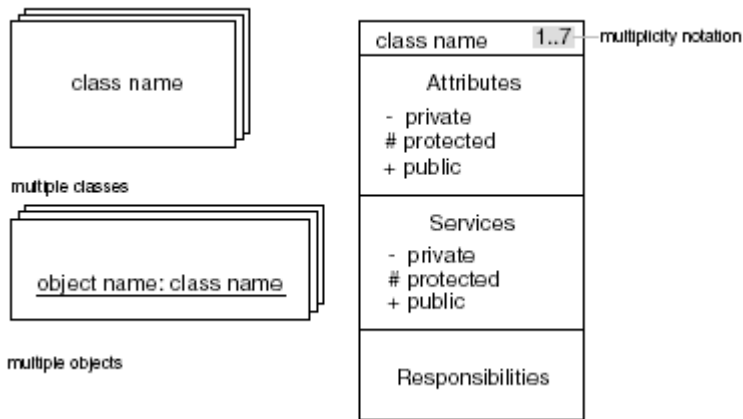
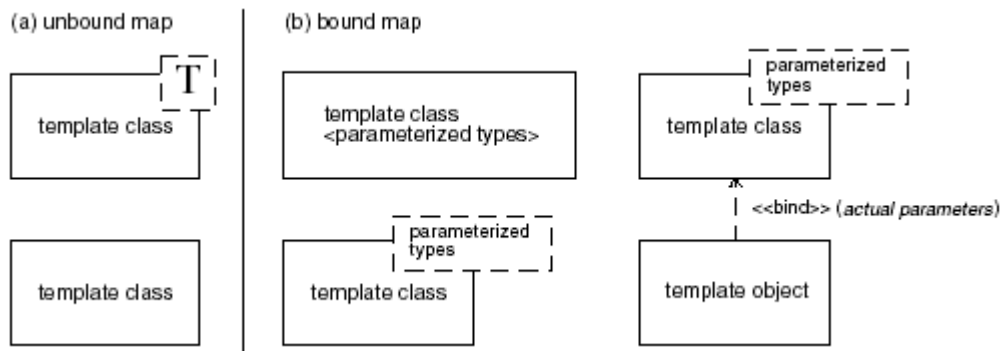


Figure A-3. The ways to represent bound or unbound template or parameterized classes.



## A.2 Interaction Diagrams

Interaction diagrams show the interaction between objects. It consists of a set of objects, their relationship, and the messages exchanged between them. Interaction diagrams include collaboration, sequence, and activity diagrams.

### A.2.1 Collaboration Diagrams

Collaboration diagrams are used to show a set of objects working together to perform some work. The collaboration in the system is a temporary cooperation between a set of objects. Collaboration diagrams can depict the organization of the collaboration or can depict the structure of the collaboration. This involves showing all the objects in the set, their links, and the messages sent and received between them.

### A.2.2 Sequence Diagrams

Sequence diagrams are used to emphasize the time ordering of messages received and sent by objects in a system.

**Figure A-4. The ways to represent an interface class. An interface class can be represented using a lollipop symbol or as a regular class displaying the <<interface>> stereotype. The relationship between the interface class and the realization of the class can also be depicted.**

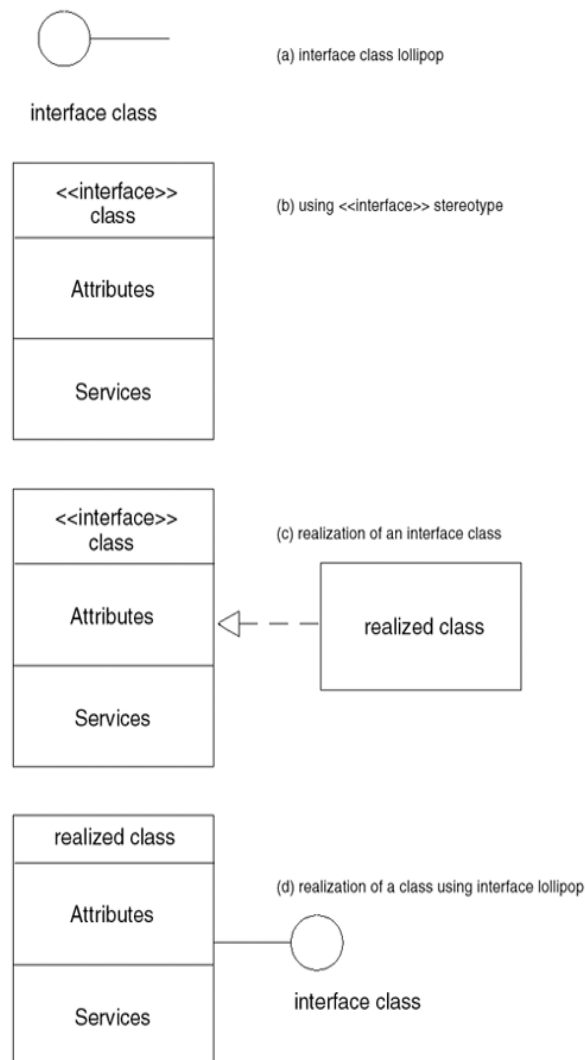
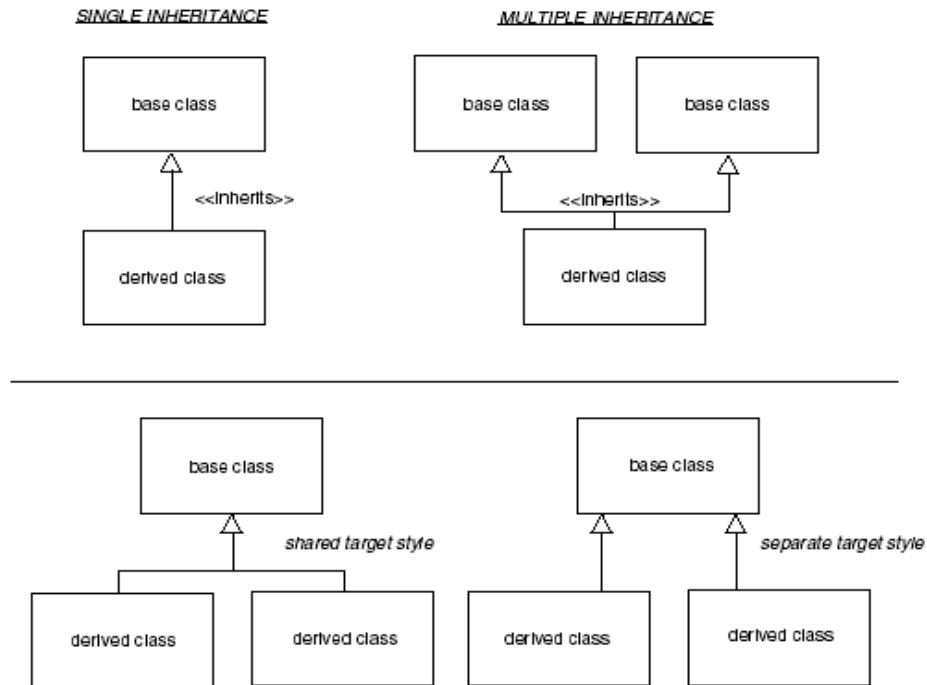


Figure A-5. The ways to represent single and multiple inheritance. There are two target styles that can be used when multiple classes are involved in a relationship: shared and separate. With the shared target style, multiple classes are tied to a single inheritance symbol that points to the target class. With the separate target style, each class has its own inheritance symbol.



### A.2.3 Activity Diagrams

Activity diagrams show the flow of control from one activity to another. Activities are actions performed by objects. Actions include processing input/output, creating or destroying objects, or performing computations. Activity diagrams are similar to flowcharts.

Figure A-6. Examples of the multiple relationships that can be depicted in a class diagram. Multiplicity notation can be used to show the number of instances between classes and objects.

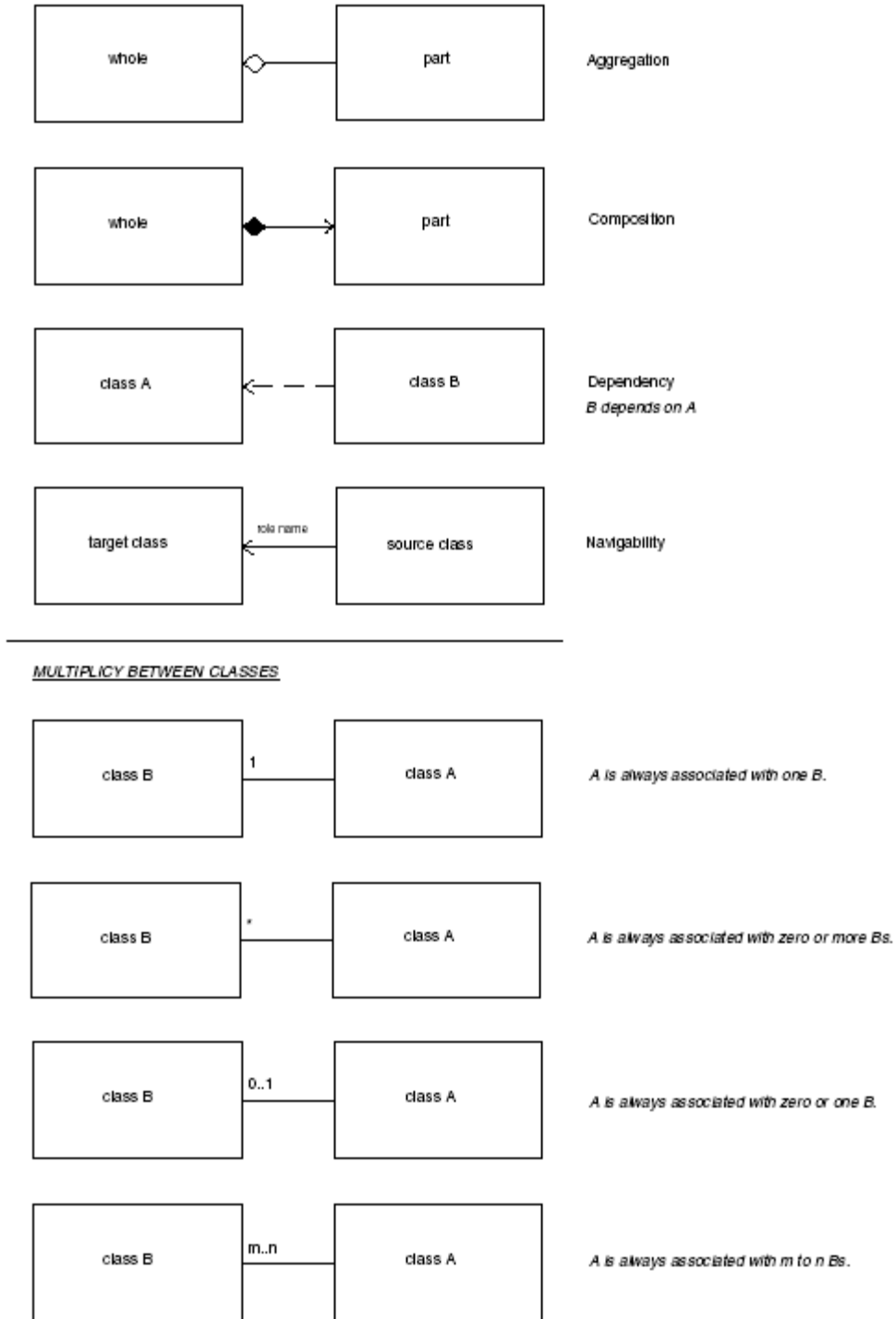


Figure A-7. A collaboration diagram showing the organization of collaborations within a system and the structural relationship of objects within a collaboration.

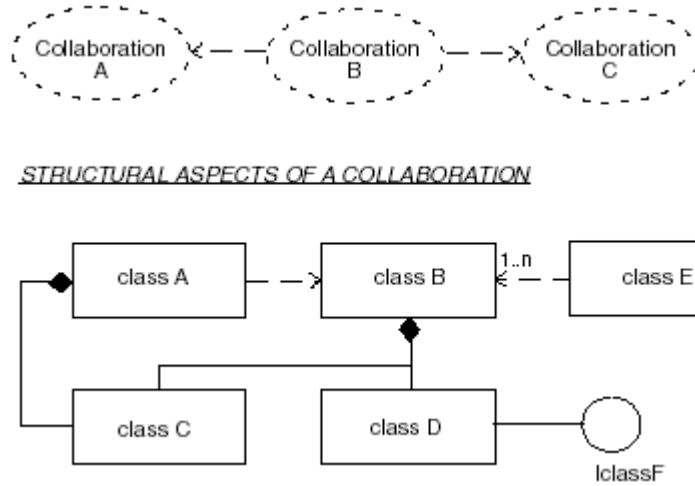


Figure A-8. A sequence diagram is used to emphasize the time ordering of messages passed between objects. The active objects are placed at the top on the x-axis of the diagram. The messages passed between the objects are placed on the y-axis of the diagram. The diagram can depict synchronous and asynchronous messaging. The time ordering of messages is demonstrated by reading the messages from top to bottom along the y-axis.

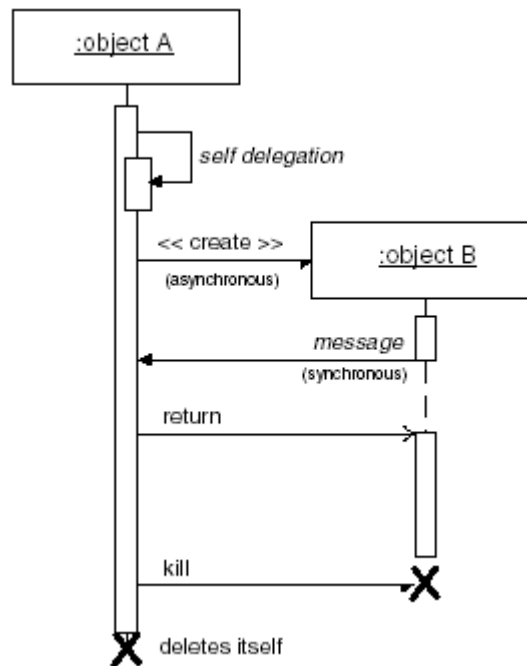
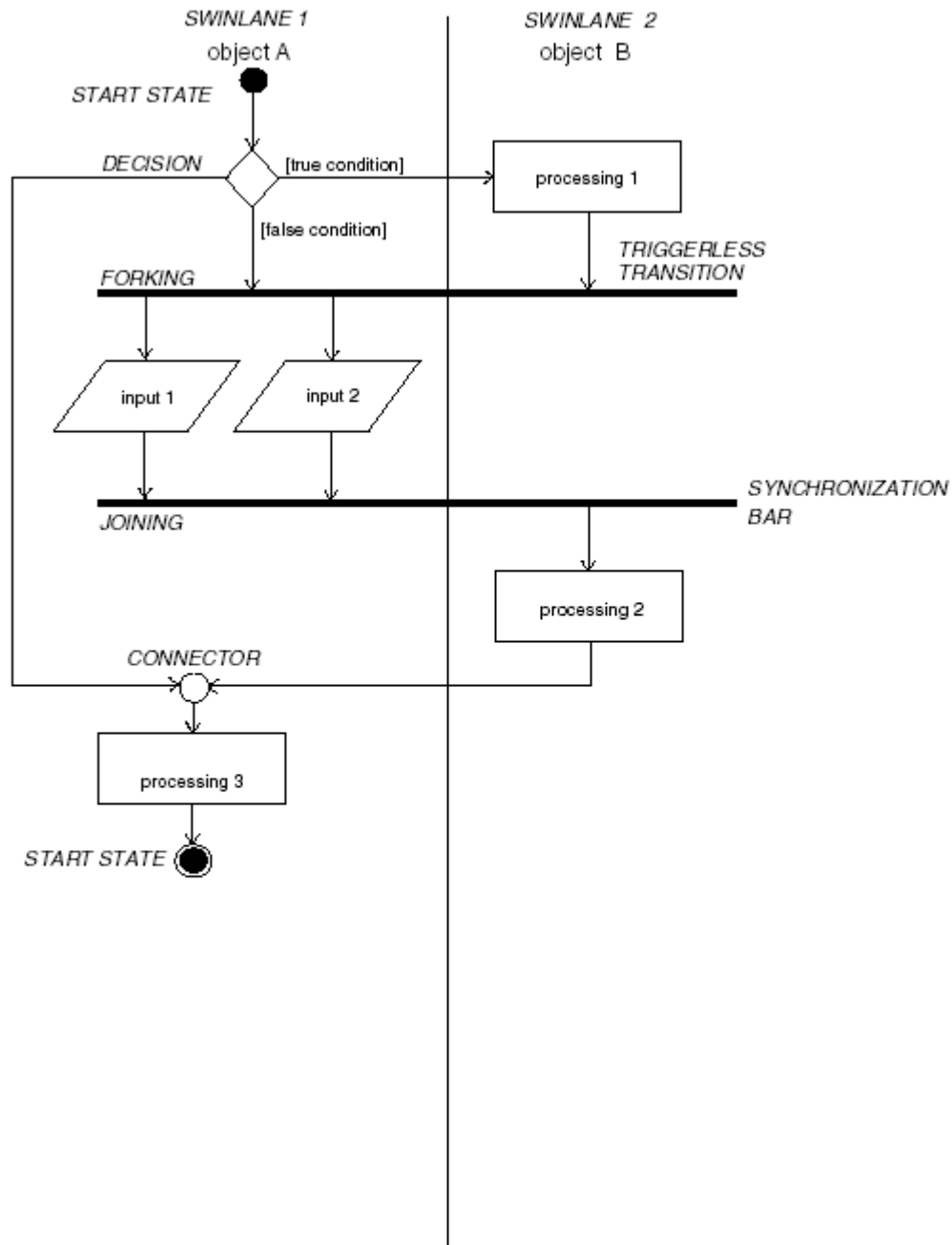




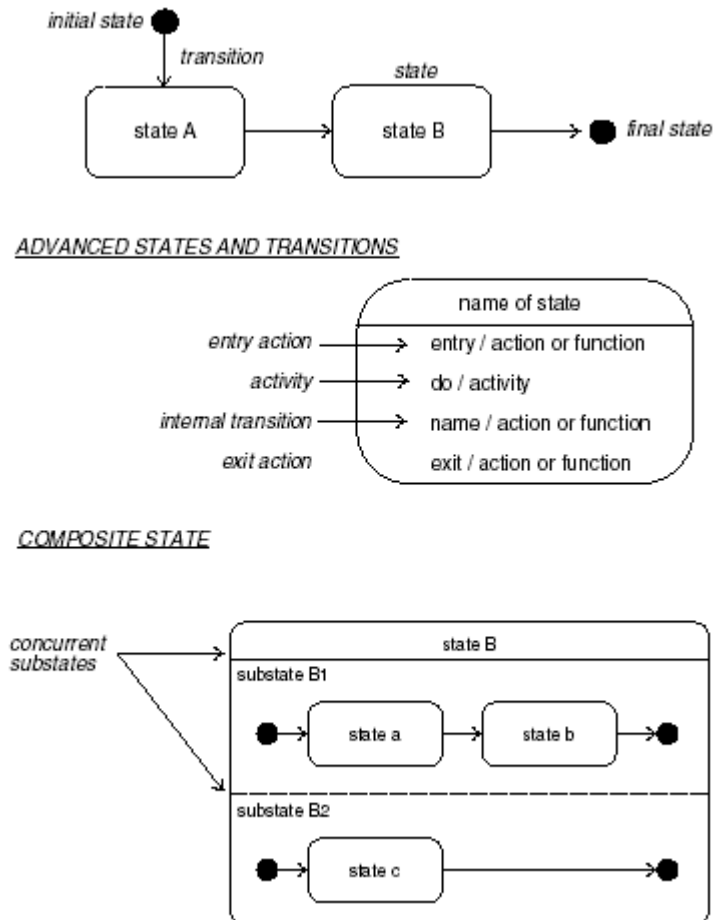
Figure A-9. Activity diagrams show the action of objects as it flows from the focus of control of one object to another. It depicts the forking of multiple flows of control (concurrency) and joining of flows of control with a synchronization bar. Swimlanes are used to show which object is performing the action. Transitions may cut across swimlanes. A synchronization bar may also cut across swimlanes, indicating multiple flows of control reside in different objects performing actions concurrently.



## A.3 State Diagrams

A state diagram is used to emphasize the state of objects and their transitions to those states. A state is a condition that an object occupies at some point in its lifetime. An object can be transformed into many different states in its lifetime. The objects transform into a state if some condition is met, some action is performed, or some event has taken place.

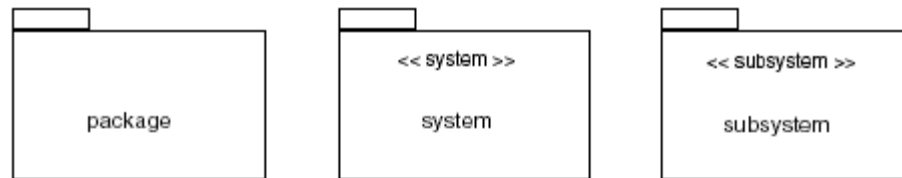
**Figure A-10.** State diagrams show the states and transitions of an object during its lifetime. A state diagram has an initial state and a final state. A state has several parts. States can also be a composite of other states or even another state diagram. Substates that execute in parallel within a single entity are called concurrent substates.



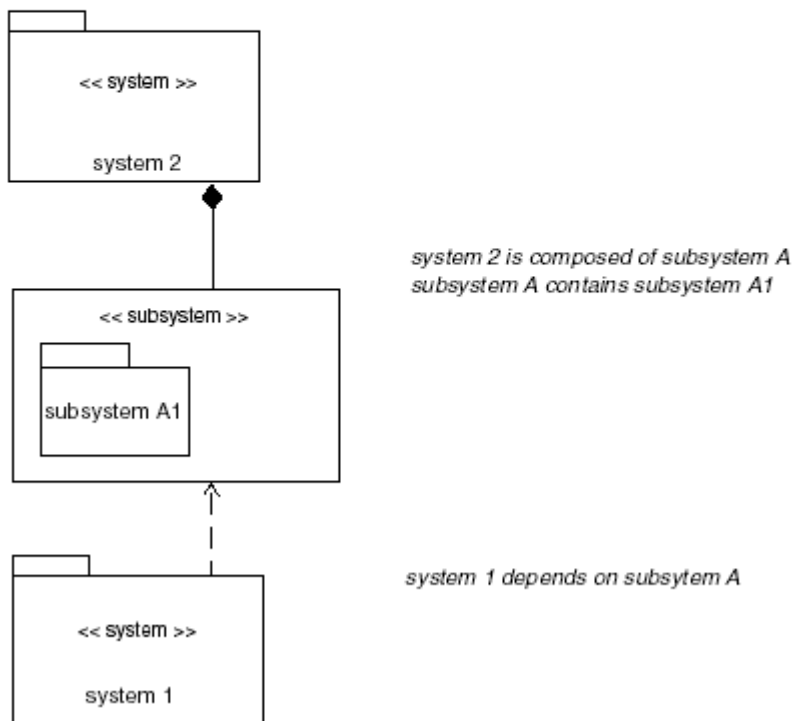
## A.4 Package Diagrams

Package diagrams are used to organize entities into groups.

Figure A-11. Package diagrams can be used to organize elements of a system. The stereotypes <<system>> or <<subsystem>> can be used. The tab on the left can hold the name of the package if the package contains other entities.



### RELATIONSHIPS BETWEEN SYSTEMS



## Bibliography

- Audi, Robert. *Action, Intention, and Reason*. Ithaca, N.Y.: Cornell University Press, 1993.
- Axford, Tom. *Concurrent Programming: Fundamental Techniques for Real-Time and Parallel Software Design*. Chichester, U.K.: John Wiley, 1989.
- Baase, Sarah. *Computer Algorithms: Introduction to Design and Analysis*. 2nd ed. Reading, Mass.: Addison-Wesley, 1988.
- Barfield, Woodrow, and Thomas A. Furnell III. *Virtual Environments and Advanced Interface Design*. New York: Oxford University Press, 1995.
- Binkley, Robert, Bronaugh, Richard, and Ausonio Marras. *Agent, Action, and Reason*. Toronto: University of Toronto Press, 1971.
- Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Boston: Addison-Wesley, 1999.
- Bowan, Howard, and John Derrick. *Formal Methods for Distributed Processing: A Survey of Object-Oriented Approaches*. New York: Cambridge University Press, 2001.
- Brewka, Gerhard, Jurgen Diz, and Kurt Konolige. *Nonmonotonic Reasoning*. Stanford, Calif.: CSLI Publications, 1997.
- Carroll, Martin D., and Margaret A. Ellis. *Designing and Coding Reusable C++*. Reading, Mass.: Addison-Wesley, 1995.
- Cassell, Justine, Joseph Sullivan, Scott Prevost, and Elizabeth Churchill. *Embodied Conversational Agents*. Cambridge, Mass.: MIT Press, 2000.
- Chellas, Brian F. *Modal Logic: An Introduction*. New York: Cambridge University Press, 1980.
- Coplien, James O. *Multi-Paradigm Design for C++*. Reading, Mass.: Addison-Wesley, 1999.
- Cormen, Thomas, Charles Leiserson, and Ronald Rivet. *Introduction to Algorithms*. Cambridge, Mass.: MIT Press, 1995.
- Englemore, Robert, and Tony Morgan. *Blackboard Systems*. Wokingham, England: Addison-Wesley, 1988.
- Garg, Vijay K. *Principles of Distributed Systems*. Norwell, Mass.: Kluwer Academic, 1996.
- Geist, Al, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sinderman. *PVM: Parallel Virtual Machine*. London, England: MIT Press, 1994.
- Goodheart, Berny, and James Cox. *The Magic Garden Explained: The Internals of Unix System V Release 4*. New York: Prentice Hall, 1994.
- Gropp, William, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference*. Vol. 2. Cambridge, Mass.: MIT Press, 1998.
- Heath, Michael T. *Scientific Computing: An Introduction Survey*. New York: McGraw-Hill.
- Henning, Michi, and Steve Vinoski. *Advanced COBRA Programming with C++*. Reading, Mass.: Addison-Wesley, 1999.
- Hintikka, Jakko, and Merrill Hintikka. *The Logic of Epistemology and the Epistemology of Logic*. Amsterdam: Kluwer Academic, 1989.
- Horty, John F. *Agency and Deontic Logic*. New York: Oxford University Press, 2001.
- Hughes, Cameron, and Tracey Hughes. *Mastering the Standard C++ Classes*. New York: John Wiley, 1990.
- Hughes, Cameron, and Tracey Hughes. *Object-Oriented Multithreading Using C++*. New York: John Wiley, 1997.
- Hughes, Cameron, and Tracey Hughes. *Linux Rapid Application Development*. Foster City, Calif.: M & T Books, 2000.
- International Standard Organization. *Information Technology: Portable Operating System Interface*. Pt. 1 System Application Program Interface. 2nd ed. Std 1003.1 ANSI/IEEE. 1996.
- Josuttis, Nicolai M. *The C++ Standard*. Boston: Addison-Wesley, 1999.
- Koeing, Andrew, and Barbara Moo. *Ruminations on C++*. Reading, Mass.: Addison-Wesley, 1997.
- Krishnamoorthy, C.S., and S. Rajeev. *Artificial Intelligence and Expert Systems for Engineers*. Boca Raton, Fla.: CRC Press, 1996.
- Lewis, Ted, Glenn Andert, Paul Calder, Erich Gamma, Wolfgang Press, Larry Rosenstein, and Kraus, Sarit. *Strategic Negotiation in Multitangent Environments*. London: MIT Press, 2001.

- Luger, George F. Artificial Intelligence. 4th ed. England: Addison-Wesley, 2002.
- Mandrioli Dino, and Carlo Ghezzi. Theoretical Foundations of Computer Science. New York: John Wiley, 1987.
- Nielsen, Michael A., and Isaac L. Chuang. Quantum Computation and Quantum Information. New York: Cambridge University Press, 2000.
- Patel, Mukesh J., Vasant Honavar, and Karthik Balakrishnan. Advances in the Evolutionary Synthesis of Intelligent Agents. Cambridge, Mass.: MIT Press, 2001.
- Picard, Rosalind. Affective Computing. London: MIT Press, 1997.
- Rescher, Nicholas, and Alasdir Urquhart. Temporal Logic. New York: Springer-Verlag, 1971.
- Robbins, Kay A., and Steven Robbins. Practical Unix Programming. Upper Saddle River, N.J.: Prentice Hall, 1996.
- Schmucker, Kurt, Ander Weinand, and John M. Vlissides. Object-Oriented Application Frameworks. Greenwich, Conn.: Manning Publications, 1995.
- Singh, Harry. Progressing to Distributed Multiprocessing. Upper Saddle River, N.J.: Prentice Hall, 1999.
- Skillicorn, David. Foundations of Parallel Programming. New York: Cambridge University Press, 1994.
- Soukup, Jiri. Taming C++: Pattern Classes and Persistence for Large Projects. Reading, Mass.: Addison-Wesley, 1994.
- Sterling, Thomas L., John Salmon, Donald J. Becker, and Daniel F. Savarese. How to Build a Bewoulf: A Guide to Implementation and Application of PC Clusters. London: MIT Press, 1999.
- Stevens, Richard W. UNIX Network Programming: Interprocess Communications. Vol. 2, 2nd ed. Upper Saddle River: Prentice Hall, 1999.
- Stroustrup, Bjarne. The Design and Evolution of C++. Reading, Mass.: Addison-Wesley, 1994.
- Subrahmanian, V.S., Piero Bonatti, Jurgen Dix, Thomas Eiter, Sarit Kraus, Fatma Ozcan, and Robert Ross. Heterogeneous Agent Systems. Cambridge, Mass.: MIT Press, 2000.
- Tel, Gerard. Introduction to Distributed Algorithms. 2nd ed. New York: Cambridge University Press, 2000.
- Thompson, William J. Computing for Scientists and Engineers. New York: John Wiley, 1992.
- Tomas, Gerald, and Christoph W. Ueberhuber. Visualization of Scientific Parallel Programming. New York: Springer-Verlag, 1994.
- Tracy, Kim W. and Peter Bouthoorn. Object-Oriented: Artificial Intelligence Using C++. New York: Computer Science Press, 1997.
- Weiss, Gerhard. Multitagent Systems. Cambridge, Mass.: MIT Press, 1999.
- Wooldridge, Michael. Reasoning About Rational Agents. London: MIT Press, 2000.

## Table of Contents

Copyright.....	2
Dedication.....	3
Preface.....	3
The Challenges.....	3
The Approach.....	4
Why C++?.....	4
Libraries for Parallel and Distributed Programming.....	5
The New Single UNIX Specification Standard.....	5
Who is This Book For?.....	5
Development Environments Supported.....	5
Ancillaries.....	6
UML Diagrams.....	6
Program Profiles.....	6

Sidebars.....	6
Testing and Code Reliability.....	6
Acknowledgments.....	6
Chapter 1. The Joys of Concurrent Programming.....	7
1.1 What is Concurrency?.....	8
1.1.1 The Two Basic Approaches to Achieving Concurrency.....	8
1.2 The Benefits of Parallel Programming.....	10
1.2.1 The Simplest Parallel Model (PRAM).....	10
1.2.2 The Simplest Parallel Classification.....	11
1.3 The Benefits of Distributed Programming.....	12
1.3.1 The Simplest Distributed Programming Models.....	12
1.3.2 The Multiagent (Peer-to-Peer) Distributed Model.....	13
1.4 The Minimal Effort Required.....	13
1.4.1 Decomposition.....	13
1.4.2 Communication.....	14
1.4.3 Synchronization.....	14
1.5 The Basic Layers of Software Concurrency.....	14
1.5.1 Concurrency at the Instruction Level.....	14
1.5.2 Concurrency at the Routine Level.....	15
1.5.3 Concurrency at the Object Level.....	15
1.5.4 Concurrency of Applications.....	15
1.6 No Keyword Support for Parallelism in C++.....	15
1.6.1 The Options for Implementing Parallelism Using C++.....	16
1.6.2 MPI Standard.....	17
1.6.3 PVM: A Standard for Cluster Programming.....	17
1.6.4 The CORBA Standard.....	17
1.6.5 Library Implementations Based on Standards.....	18
1.7 Programming Environments for Parallel and Distributed Programming.....	18
Summary—Toward Concurrency.....	19
Chapter 2. The Challenges of Parallel and Distributed Programming.....	20
2.1 The Big Paradigm Shift.....	20
2.2 Coordination Challenges.....	23
Problem #1 Data Race.....	23
Problem #2 Indefinite Postponement.....	23
Problem #3 Deadlock.....	24
Problem #4 Communication Difficulties.....	25
2.3 Sometimes Hardware Fails and Software Quits.....	26
2.4 Too Much Parallelization or Distribution Can Have Negative Consequences.....	27
2.5 Selecting a Good Architecture Requires Research.....	27
2.6 Different Techniques for Testing and Debugging are Required.....	29
2.7 The Parallel or Distributed Design Must Be Communicated.....	29
Summary.....	30
Chapter 3. Dividing C++ Programs into Multiple Tasks.....	31
3.1 Process: A Definition.....	31
3.1.1 Two Kinds of Processes.....	32
3.1.2 Process Control Block.....	32
3.2 Anatomy of a Process.....	33
3.3 Process States.....	35
3.4 Process Scheduling.....	38
3.4.1 Scheduling Policy.....	38

3.4.2 Using the ps Utility.....	39
Synopsis.....	41
3.4.3 Setting and Returning the Process Priority.....	43
Synopsis.....	43
Synopsis.....	44
3.5 Context Switching.....	45
3.6 Creating a Process.....	46
3.6.1 Parent–Child Process Relationship.....	46
Synopsis.....	49
3.6.2 Using the fork() Function Call.....	50
Synopsis.....	50
3.6.3 Using the exec Family of System Calls.....	50
Synopsis.....	52
Synopsis.....	53
Synopsis.....	54
3.6.4 Using system() to Spawn Processes.....	54
Synopsis.....	55
3.6.5 The POSIX Functions for Spawning Processes.....	55
Synopsis.....	55
3.6.6 Identifying the Parent and Child with Process Management Functions.....	60
Synopsis.....	60
3.7 Terminating a Process.....	61
3.7.1 The exit(), kill() and abort() Calls.....	61
Synopsis.....	61
Synopsis.....	62
3.8 Process Resources.....	63
S 3.1 Resource Allocation Graph.....	63
3.8.1 Types of Resources.....	64
3.8.2 POSIX Functions to Set Resource Limits.....	65
Synopsis.....	65
3.9 What are Asynchronous and Synchronous Processes?.....	68
3.9.1 Synchronous and Asynchronous Processes Created with fork(), exec(), system(), and posix_spawn() Functions.....	70
3.9.2 The wait() Function Call.....	70
Synopsis.....	70
3.10 Dividing the Program into Tasks.....	73
3.10.1 Processes Along Function and Object Lines.....	79
Summary.....	80
Chapter 4. Dividing C++ Programs into Multiple Threads.....	81
4.1 Threads: A Definition.....	82
4.1.1 Thread Context Requirements.....	82
4.1.2 Threads and Processes: A Comparison.....	83
4.1.3 Advantages of Threads.....	84
4.1.4 Disadvantages of Threads.....	86
4.2 The Anatomy of a Thread.....	88
4.2.1 Thread Attributes.....	89
4.3 Thread Scheduling.....	90
4.3.1 Thread States.....	91
4.3.2 Scheduling and Thread Contention Scope.....	92
4.3.3 Scheduling Policy and Priority.....	93

4.4 Thread Resources.....	95
4.5 Thread Models.....	95
4.5.1 Delegation Model.....	96
4.5.2 Peer-to-Peer Model.....	97
4.5.3 Pipeline Model.....	98
4.5.4 Producer-Consumer Model.....	99
4.5.5 SPMD and MPMD for Threads.....	99
4.6 Introduction to the Pthread Library.....	100
4.7 The Anatomy of a Simple Threaded Program.....	101
4.7.1 Compiling and Linking Multithreaded Programs.....	102
4.8 Creating Threads.....	103
Synopsis.....	103
Program Profile 4.1.....	105
4.8.1 Getting the Thread Id.....	105
Synopsis.....	106
4.8.2 Joining Threads.....	106
Synopsis.....	106
4.8.3 Creating Detached Threads.....	107
Synopsis.....	107
4.8.4 Using the Pthread Attribute Object.....	107
Synopsis.....	108
Synopsis.....	108
4.9 Managing Threads.....	109
4.9.1 Terminating Threads.....	109
Synopsis.....	110
Synopsis.....	110
Synopsis.....	111
Synopsis.....	112
Program Profile 4.2.....	114
Synopsis.....	117
4.9.2 Managing the Thread's Stack.....	118
Synopsis.....	119
Synopsis.....	119
Synopsis.....	120
4.9.3 Setting Thread Scheduling and Priorities.....	120
Synopsis.....	121
Synopsis.....	122
Synopsis.....	123
Synopsis.....	125
4.9.4 Using sysconf().....	125
Synopsis.....	125
4.9.5 Managing a Critical Section.....	127
4.10 Thread Safety and Libraries.....	132
4.11 Dividing Your Program into Multiple Threads.....	134
4.11.1 Using the Delegation Model.....	134
4.11.2 Using the Peer-to-Peer Model.....	138
4.11.3 Using the Pipeline Model.....	139
4.11.4 Using the Producer-Consumer Model.....	140
4.11.5 Creating Multithreaded Objects.....	141
Summary.....	142



Chapter 5. Synchronizing Concurrency between Tasks.....	143
5.1 Coordinating Order of Execution.....	144
5.1.1 Relationships between Synchronized Tasks.....	144
5.1.2 Start-to-Start (SS) Relationship.....	145
5.1.3 Finish-to-Start (FS) Relationship.....	145
5.1.4 Start-to-Finish Relationship.....	145
5.1.5 Finish-to-Finish Relationship.....	146
5.2 Synchronizing Access to Data.....	146
5.2.1 PRAM Model.....	147
5.3 What are Semaphores?.....	149
5.3.1 Semaphore Operations.....	149
5.3.2 Mutex Semaphores.....	150
5.3.3 Read–Write Locks.....	157
5.3.4 Condition Variables.....	160
5.4 Synchronization: An Object-Oriented Approach.....	165
Summary.....	165
Chapter 6. Adding Parallel Programming Capabilities to C++ Through the PVM.....	166
6.1 The Classic Parallelism Models Supported by PVM.....	167
6.2 The PVM Library for C++.....	168
Program Profile 6.1.....	169
Program Profile 6.2.....	170
6.2.1 Compiling and Linking a C++/PVM Program.....	171
6.2.2 Executing a PVM Program as a Standalone.....	171
6.2.3 A PVM Preliminary Requirements Checklist.....	173
6.2.4 Combining the C++ Runtime Library and the PVM Library.....	177
6.2.5 Approaches to Using PVM Tasks.....	178
S 6.1. Combination Notation.....	185
6.3 The Basic Mechanics of the PVM.....	186
6.3.1 Process Management and Control Routines.....	187
Synopsis.....	187
6.3.2 Message Packing and Sending.....	188
Synopsis.....	190
Synopsis.....	191
Synopsis.....	193
6.4 Accessing Standard Input (stdin) and Standard Output (stdout) within PVM Tasks.....	194
6.4.1 Retrieving Standard Output (cout) from a Child Task.....	194
Summary.....	195
Chapter 7. Error Handling, Exceptions, and Software Reliability.....	196
7.1 What is Software Reliability?.....	197
7.2 Failures in Software Layers and Hardware Components.....	198
7.3 Definitions of Defects Depend on Software Specifications.....	199
7.4 Recognizing Where to Handle Defects versus Where to Handle Exceptions.....	200
7.5 Software Reliability: A Simple Plan.....	202
7.5.1 Plan A: The Resumption Model, Plan B: The Termination Model.....	203
7.6 Using Map Objects in Error Handling.....	203
7.7 Exception Handling Mechanisms in C++.....	206
7.7.1 The Exception Classes.....	207
7.8 Event Diagrams, Logic Expressions, and Logic Diagrams.....	210
Summary.....	212
Chapter 8. Distributed Object-Oriented Programming in C++.....	213

8.1 Decomposition and Encapsulation of the Work.....	215
8.1.1 Communication between Distributed Objects.....	217
8.1.2 Synchronization of the Interaction between the Local and the Remote Objects.....	217
8.1.3 Error and Exception Handling in the Distributed Environment.....	217
8.2 Accessing Objects in Other Address Spaces.....	218
8.2.1 IOR Access to Remote Objects.....	219
8.2.2 ORBS (Object Request Brokers).....	220
8.2.3 Interface Definition Language (IDL): A Closer Look at CORBA Objects.....	223
8.3 The Anatomy of a Basic CORBA Consumer.....	227
8.4 The Anatomy of a CORBA Producer.....	229
8.5 The Basic Blueprint of a CORBA Application.....	230
Program Profile 8.1.....	231
Program Profile 8.2.....	232
8.5.1 The IDL Compiler.....	232
8.5.2 Obtaining IOR for Remote Objects.....	233
8.6 The Naming Service.....	235
S 8.1. Semantic Networks.....	237
8.6.1 Using the Naming Service and Creating Naming Contexts.....	238
8.6.2 A Name Service Consumer/Client.....	240
8.7 A Closer Look at Object Adapters.....	243
8.8 Implementation and Interface Repositories.....	245
8.9 Simple Distributed Web Services Using CORBA.....	245
8.10 The Trading Service.....	246
8.11 The Client/Server Paradigm.....	248
Summary.....	248
Chapter 9. SPMD and MPMD Using Templates and the MPI.....	250
9.1 Work Breakdown Structure for the MPI.....	251
9.1.1 Differentiating Tasks by Rank.....	251
9.1.2 Grouping Tasks by Communicators.....	253
9.1.3 The Anatomy of an MPI Task.....	253
9.2 Using Template Functions to Represent MPI Tasks.....	257
9.2.1 Instantiating Templates and SPMD (Datatypes).....	257
9.2.2 Using Polymorphism to Implement MPMD.....	258
9.2.3 Adding MPMD with Function Objects.....	262
9.3 Simplifying MPI Communications.....	264
9.3.1 Overloading the << and >> Operators for MPI Communication.....	267
Summary.....	270
Chapter 10. Visualizing Concurrent and Distributed System Design.....	270
10.1 Visualizing Structures.....	271
10.1.1 Classes and Objects.....	271
10.1.2 The Relationship between Classes and Objects.....	279
10.1.3 The Organization of Interactive Objects.....	285
10.2 Visualizing Concurrent Behavior.....	286
10.2.1 Collaborating Objects.....	286
10.2.2 Message Sequences between Objects.....	290
10.2.3 The Activities of Objects.....	292
10.2.4 State Machines.....	295
10.2.5 Distributed Objects.....	299
10.3 Visualizing the Whole System.....	301
10.3.1 Visualizing Deployment of Systems.....	301

10.3.2 The Architecture of a System.....	302
Summary.....	304
Chapter 11. Designing Components That Support Concurrency.....	305
11.1 Taking Advantage of Interface Classes.....	307
Synopsis.....	307
11.2 A Closer Look at Object-Oriented Mutual Exclusion and Interface Classes.....	312
11.2.1 Semi-Fat Interfaces that Support Concurrency.....	312
Synopsis.....	314
Synopsis.....	316
11.3 Maintaining the Stream Metaphor.....	318
11.3.1 Overloading the <<, >> Operators for PVM Streams.....	320
11.4 User-Defined Classes Designed to Work with PVM Streams.....	323
11.5 Object-Oriented Pipes and fifos as Low-Level Building Blocks.....	325
Synopsis.....	328
11.5.1 Connecting Pipes to iostream Objects Using File Descriptors.....	328
Program Profile 11.1.....	330
11.5.2 Accessing Anonymous Pipes Using the ostream_iterator.....	332
Program Profile 11.2.....	335
Program Profile 11.2b.....	337
11.5.3 fifos (Named Pipes), iostreams, and the ostream_iterator Classes.....	337
Program Profile 11.3a.....	339
Program Profile 11.3b.....	341
11.6 Framework Classes Components for Concurrency.....	342
Summary.....	346
Chapter 12. Implementing Agent-Oriented Architectures.....	347
12.1 What are Agents?.....	347
12.1.1 Agents: A First-Cut Definition.....	348
12.1.2 Types of Agents.....	348
12.1.3 What is the Difference between Objects and Agents?.....	349
S 12.1. Deduction, Induction, and Abduction.....	350
12.2 What is Agent-Oriented Programming?.....	352
12.2.1 Why Agents Work for Distributed Programming.....	352
12.2.2 Agents and Parallel Programming.....	353
12.3 Basic Agent Components.....	355
12.3.1 Cognitive Data Structures.....	355
12.4 Implementing Agents in C++.....	360
12.4.1 Proposition Datatypes and Belief Structures.....	361
12.4.2 The Agent Class.....	365
12.4.3 Simple Autonomy.....	374
12.5 Multiagent Systems.....	376
Summary.....	376
Chapter 13. Blackboard Architectures Using PVM, Threads, and C++ Components.....	377
13.1 The Blackboard Model.....	378
13.2 Approaches to Structuring the Blackboard.....	380
13.3 The Anatomy of a Knowledge Source.....	381
13.4 The Control Strategies for Blackboards.....	383
13.5 Implementing the Blackboard Using CORBA Objects.....	384
13.5.1 The CORBA Blackboard: An Example.....	385
13.5.2 The Implementation of the black_board Interface Class.....	386
13.5.3 Spawning the Knowledge Sources in the Blackboard's Constructor.....	388

13.6 Implementing the Blackboard Using Global Objects.....	398
13.7 Activating Knowledge Sources Using Pthreads.....	401
Summary.....	403
Appendix A. Diagrams.....	403
A.1 Class and Object Diagrams.....	403
A.2 Interaction Diagrams.....	405
A.2.1 Collaboration Diagrams.....	405
A.2.2 Sequence Diagrams.....	405
A.2.3 Activity Diagrams.....	407
A.3 State Diagrams.....	410
A.4 Package Diagrams.....	411
Bibliography.....	412