CHAPTER 17

NETWORK PROTOCOLS

Learning Objectives

After studying this chapter, you should be able to:

- Explain the motivation for organizing communication functions into a layered protocol architecture.
- Describe the TCP/IP protocol architecture.
- Understand the purpose of the Sockets facility and how to use it.
- Describe the networking features in Linux.
- Understand how TFTP works.

With the increasing availability of inexpensive yet powerful personal computers and servers, there has been an increasing trend toward distributed data processing (DDP), in which processors, data, and other aspects of a data processing system may be dispersed within an organization. A DDP system involves a partitioning of the computing function and may also involve a distributed organization of databases, device control, and interaction (network) control.

In many organizations, there is heavy reliance on personal computers coupled with servers. Personal computers are used to support a variety of user-friendly applications, such as word processing, spreadsheet, and presentation graphics. The servers house the corporate database plus sophisticated database management and information systems software. Linkages are needed among the personal computers and between each personal computer and the server. Various approaches are in common use, ranging from treating the personal computer as a simple terminal to implementing a high degree of integration between personal computer applications and the server database.

These application trends have been supported by the evolution of distributed capabilities in the operating system and supporting utilities. A spectrum of distributed capabilities has been explored:

- **Communications architecture:** This is software that supports a group of networked computers. It provides support for distributed applications, such as electronic mail, file transfer, and remote terminal access. However, each computer retains a distinct identity to the user and to the applications, which must communicate with other computers by explicit reference. Each computer has its own separate operating system, and a heterogeneous mix of computers and operating systems is possible, as long as all machines support the same communications architecture. The most widely used communications architecture is the TCP/IP protocol suite, examined in this chapter.
- **Network operating system:** This is a configuration in which there is a network of application machines, usually single-user workstations and one or more "server" machines. The server machines provide networkwide services or applications, such as file storage and printer management. Each computer has its own private operating system. The network operating system is simply an adjunct to the local operating system that allows application machines to interact with server machines. The user is aware that there are multiple independent computers and must deal with them explicitly. Typically, a common communications architecture is used to support these network applications.

- **Distributed operating system:** A common operating system shared by a network of computers. It looks to its users like an ordinary centralized operating system but provides the user with transparent access to the resources of a number of machines. A distributed operating system may rely on a communications architecture for basic communications functions; more commonly, a stripped-down set of communications functions is incorporated into the operating system to provide efficiency.

The technology of the communications architecture is well-developed and is supported by all vendors. Network operating systems are a more recent phenomena, but a number of commercial products exist. The leading edge of research and development for distributed systems is in the area of distributed operating systems. Although some commercial systems have been introduced, fully functional distributed operating systems are still at the experimental stage.

In this chapter and the next, we will provide a survey of distributed processing capabilities. This chapter focuses on the underlying network protocol software.

17.1 THE NEED FOR A PROTOCOL ARCHITECTURE

When computers, terminals, and/or other data processing devices exchange data, the procedures involved can be quite complex. Consider, for example, the transfer of a file between two computers. There must be a data path between the two computers, either directly or via a communication network. But more is needed. Typical tasks to be performed include the following:

1. The source system must either activate the direct data communication path or inform the communication network of the identity of the desired destination system.
2. The source system must ascertain that the destination system is prepared to receive data.
3. The file transfer application on the source system must ascertain that the file management program on the destination system is prepared to accept and store the file for this particular user.
4. If the file formats or data representations used on the two systems are incompatible, one or the other system must perform a format translation function.

The exchange of information between computers for the purpose of cooperative action is generally referred to as *computer communications*. Similarly, when two or more computers are interconnected via a communication network, the set of computer stations is referred to as a *computer network*. Because a similar level of cooperation is required between a terminal and a computer, these terms are often used when some of the communicating entities are terminals.

In discussing computer communications and computer networks, two concepts are paramount:

- Protocols
- Computer communications architecture, or protocol architecture

A **protocol** is used for communication between entities in different systems. The terms *entity* and *system* are used in a very general sense. Examples of entities are user application programs, file transfer packages, database management systems, electronic mail facilities, and terminals. Examples of systems are computers, terminals, and remote sensors. Note in some cases the entity and the system in which it resides are coextensive (e.g., terminals). In general, an entity is anything capable of sending or receiving information, and a system is a physically distinct object that contains one or more entities. For two entities to communicate successfully, they must "speak the same language." What is communicated, how it is communicated, and when it is communicated must conform to mutually agreed conventions between the entities involved. The conventions are referred to as a protocol, which may be defined as a set of rules governing the exchange of data between two entities. The key elements of a protocol are as follows:

- **Syntax:** Includes such things as data format and signal levels
- **Semantics:** Includes control information for coordination and error handling
- **Timing:** Includes speed matching and sequencing

Appendix 17A provides a specific example of a protocol, the Internet standard Trivial File Transfer Protocol (TFTP).

Having introduced the concept of a protocol, we can now introduce the concept of a **protocol architecture**. It is clear there must be a high degree of cooperation between the two computer systems. Instead of implementing the logic for this as a single module, the task is broken up into subtasks, each of which is implemented separately. As an example, Figure 17.1 suggests the way in which a file transfer facility could be implemented. Three modules are used. Tasks 3 and 4 in the preceding list could be performed by a file transfer module. The two modules on the two systems exchange files and commands. However, rather than requiring the file transfer module to deal with the details of actually transferring data and commands, the file transfer modules each rely on a communications service module. This module is responsible for making sure the file transfer commands and

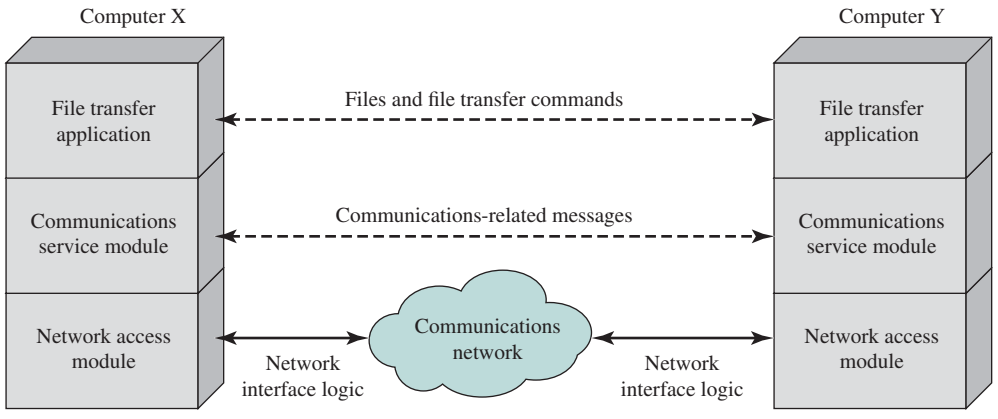


Figure 17.1 A Simplified Architecture for File Transfer

data are reliably exchanged between systems. The manner in which a communications service module functions will be explored subsequently. Among other things, this module would perform task 2. Finally, the nature of the exchange between the two communications service modules is independent of the nature of the network that interconnects them. Therefore, rather than building details of the network interface into the communications service module, it makes sense to have a third module, a network access module, that performs task 1 by interacting with the network.

To summarize, the file transfer module contains all the logic that is unique to the file transfer application, such as transmitting passwords, file commands, and file records. These files and commands must be transmitted reliably. However, the same sorts of reliability requirements are relevant to a variety of applications (e.g., electronic mail, document transfer). Therefore, these requirements are met by a separate communications service module that can be used by a variety of applications. The communications service module is concerned with assuring that the two computer systems are active and ready for data transfer, and for keeping track of the data that are being exchanged to assure delivery. However, these tasks are independent of the type of network that is being used. Therefore, the logic for actually dealing with the network is put into a separate network access module. If the network to be used is changed, only the network access module is affected.

Thus, instead of a single module for performing communications, there is a structured set of modules that implements the communications function. That structure is referred to as a protocol architecture. An analogy might be useful at this point. Suppose an executive in office X wishes to send a document to an executive in office Y. The executive in X prepares the document and perhaps attaches a note. This corresponds to the actions of the file transfer application in Figure 17.1. Then the executive in X hands the document to a secretary or administrative assistant (AA). The AA in X puts the document in an envelope and puts Y's address and X's return address on the outside. Perhaps the envelope is also marked "confidential." The AA's actions correspond to the communications service module in Figure 17.1. The AA in X then gives the package to the shipping department. Someone in the shipping department decides how to send the package: mail, UPS, or express courier. The shipping department attaches the appropriate postage or shipping documents to the package and ships it out. The shipping department corresponds to the network access module of Figure 17.1. When the package arrives at Y, a similar layered set of actions occurs. The shipping department at Y receives the package and delivers it to the appropriate AA or secretary based on the name on the package. The AA opens the package and hands the enclosed document to the executive to whom it is addressed.

17.2 THE TCP/IP PROTOCOL ARCHITECTURE

The TCP/IP protocol architecture is a result of protocol research and development conducted on the experimental packet-switched network, ARPANET, funded by the Defense Advanced Research Projects Agency (DARPA), and is generally referred to as the TCP/IP protocol suite. This protocol suite consists of a large collection of

protocols that have been issued as Internet standards by the Internet Activities Board (IAB). Appendix L provides a discussion of Internet standards.

TCP/IP Layers

In general terms, computer communications can be said to involve three agents: applications, computers, and networks. Examples of applications include file transfer and electronic mail. The applications with which we are concerned here are distributed applications that involve the exchange of data between two computer systems. These applications and others execute on computers that can often support multiple simultaneous applications. Computers are connected to networks, and the data to be exchanged are transferred by the network from one computer to another. Thus, the transfer of data from one application to another involves first getting the data to the computer in which the application resides, then getting the data to the intended application within the computer.

There is no official TCP/IP protocol model. However, based on the protocol standards that have been developed, we can organize the communication task for TCP/IP into five relatively independent layers, from bottom to top:

- Physical layer
- Network access layer
- Internet layer
- Host-to-host, or transport layer
- Application layer

The **physical layer** covers the physical interface between a data transmission device (e.g., workstation, computer) and a transmission medium or network. This layer is concerned with specifying the characteristics of the transmission medium, the nature of the signals, the data rate, and related matters.

The **network access layer** is concerned with the exchange of data between an end system (server, workstation, etc.) and the network to which it is attached. The sending computer must provide the network with the address of the destination computer, so the network may route the data to the appropriate destination. The sending computer may wish to invoke certain services, such as priority, that might be provided by the network. The specific software used at this layer depends on the type of network to be used; different standards have been developed for circuit switching, packet switching (e.g., frame relay), LANs (e.g., Ethernet), and others. Thus, it makes sense to separate those functions having to do with network access into a separate layer. By doing this, the remainder of the communications software, above the network access layer, need not be concerned about the specifics of the network to be used. The same higher-layer software should function properly regardless of the particular network to which the computer is attached.

The network access layer is concerned with access to and routing data across a network for two end systems attached to the same network. In those cases where two devices are attached to different networks, procedures are needed to allow data to traverse multiple interconnected networks. This is the function of the Internet layer. The **Internet Protocol (IP)** is used at this layer to provide the routing function across multiple networks. This protocol is implemented not only in the end systems but also

in routers. A **router** is a processor that connects two networks and whose primary function is to relay data from one network to the other on a route from the source to the destination end system.

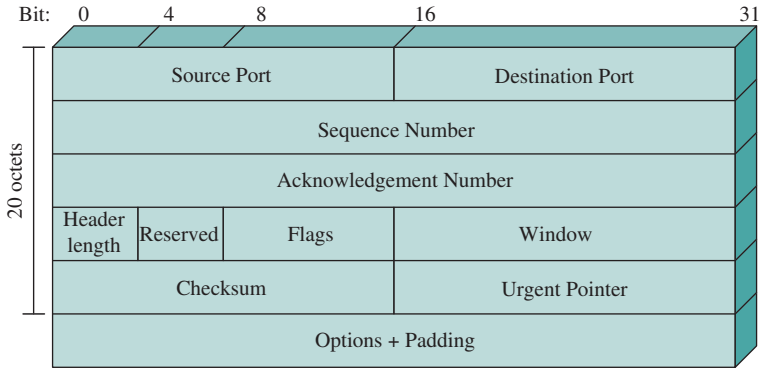
Regardless of the nature of the applications that are exchanging data, there is usually a requirement that data be exchanged reliably. That is, we would like to be assured that all the data arrive at the destination application, and that the data arrive in the same order in which they were sent. As we shall see, the mechanisms for providing reliability are essentially independent of the nature of the applications. Thus, it makes sense to collect those mechanisms in a common layer shared by all applications; this is referred to as the host-to-host layer, or **transport layer**. The Transmission Control Protocol (TCP) is the most commonly used protocol to provide this functionality.

Finally, the **application layer** contains the logic needed to support the various user applications. For each different type of application, such as file transfer, a separate module is needed that is peculiar to that application.

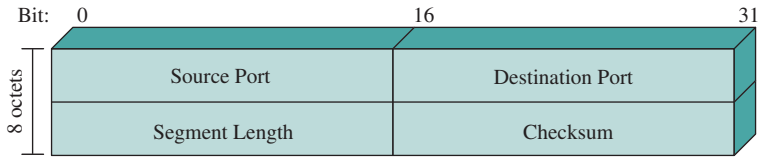
TCP and UDP

For most applications running as part of the TCP/IP protocol architecture, the transport layer protocol is TCP. TCP provides a reliable connection for the transfer of data between applications. A connection is simply a temporary logical association between two entities in different systems. For the duration of the connection, each entity keeps track of segments coming and going to the other entity, in order to regulate the flow of segments and to recover from lost or damaged segments.

Figure 17.2a shows the header format for TCP, which is a minimum of 20 octets, or 160 bits. The Source Port and Destination Port fields identify the applications at



(a) TCP Header



(b) UDP Header

Figure 17.2 TCP and UDP Headers

the source and destination systems that are using this connection. The Sequence Number, Acknowledgment Number, and Window fields provide flow control and error control. The checksum is a 16-bit code based on the contents of the segment used to detect errors in the TCP segment.

In addition to TCP, there is one other transport-level protocol that is in common use as part of the TCP/IP protocol suite: the User Datagram Protocol (UDP). UDP does not guarantee delivery, preservation of sequence, or protection against duplication. UDP enables a process to send messages to other processes with a minimum of protocol mechanism. Some transaction-oriented applications make use of UDP; one example is SNMP (Simple Network Management Protocol), the standard network management protocol for TCP/IP networks. Because it is connectionless, UDP has very little to do. Essentially, it adds a port addressing capability to IP. This is best seen by examining the UDP header, shown in Figure 17.2b.

IP and IPv6

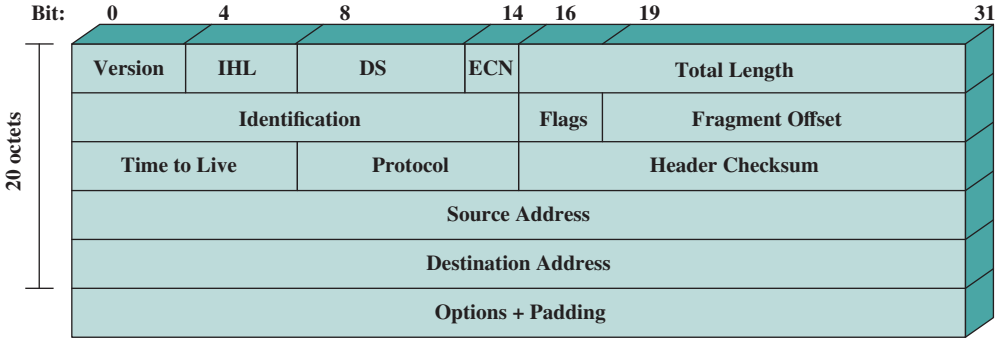
For decades, the keystone of the TCP/IP protocol architecture has been IP. Figure 17.3a shows the IP header format, which is a minimum of 20 octets, or 160 bits. The header, together with the segment from the transport layer, forms an IP-level block referred to as an IP datagram or an IP packet. The header includes 32-bit source and destination addresses. The Header Checksum field is used to detect errors in the header to avoid misdelivery. The Protocol field indicates whether TCP, UDP, or some other higher-layer protocol is using IP. The ID, Flags, and Fragment Offset fields are used in the fragmentation and reassembly process, in which a single IP datagram is divided into multiple IP datagrams on transmission then reassembled at the destination.

In 1995, the Internet Engineering Task Force (IETF), which develops protocol standards for the Internet, issued a specification for a next-generation IP, known then as IPng. This specification was turned into a standard in 1996 known as IPv6. IPv6 provides a number of functional enhancements over the existing IP, designed to accommodate the higher speeds of today's networks and the mix of data streams, including graphic and video, which are becoming more prevalent. But the driving force behind the development of the new protocol was the need for more addresses. The current IP uses a 32-bit address to specify a source or destination. With the explosive growth of the Internet and of private networks attached to the Internet, this address length became insufficient to accommodate all the systems needing addresses. As Figure 17.3b shows, IPv6 includes 128-bit source and destination address fields.

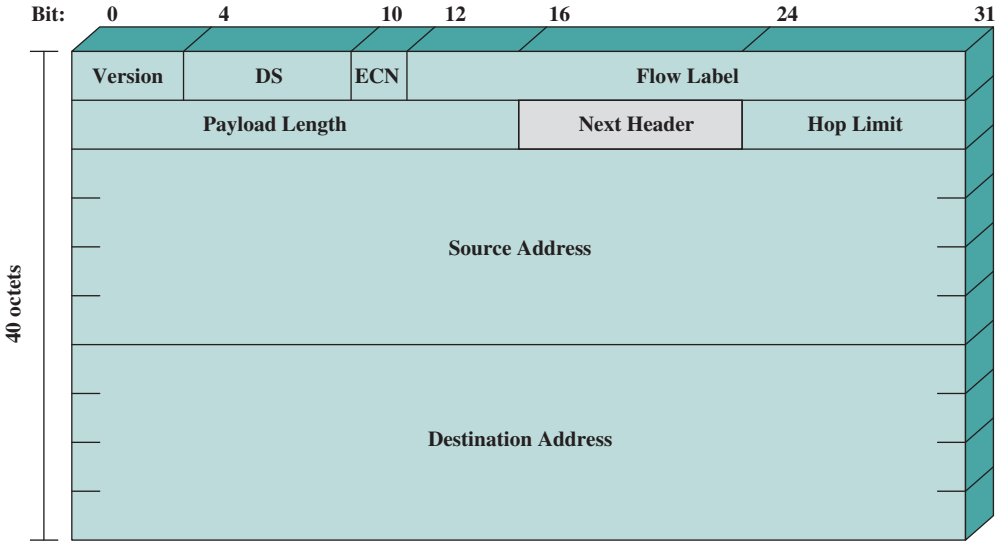
Ultimately, all installations using TCP/IP are expected to migrate from the current IP to IPv6, but this process will take many years, if not decades.

Operation of TCP/IP

Figure 17.4 indicates how these protocols are configured for communications. Some sort of network access protocol, such as the Ethernet logic, is used to connect a computer to a network. This protocol enables the host to send data across the network to another host or, in the case of a host on another network, to a router. IP is implemented in all end systems and routers. It acts as a relay to move a block of data from



(a) IPv4 Header



(b) IPv6 Header

DS = Differentiated services field
ECN = Explicit congestion notification field

Note: The 8-bit DS/ECN fields were formerly known as the Type of Service field in the IPv4 header and the Traffic Class field in the IPv6 header.

Figure 17.3 IP Headers

one host, through one or more routers, to another host. TCP is implemented only in the end systems; it keeps track of the blocks of data being transferred to assure that all are delivered reliably to the appropriate application.

For successful communication, every entity in the overall system must have a unique address. In fact, two levels of addressing are needed. Each host on a network must have a unique global Internet address; this allows the data to be delivered to

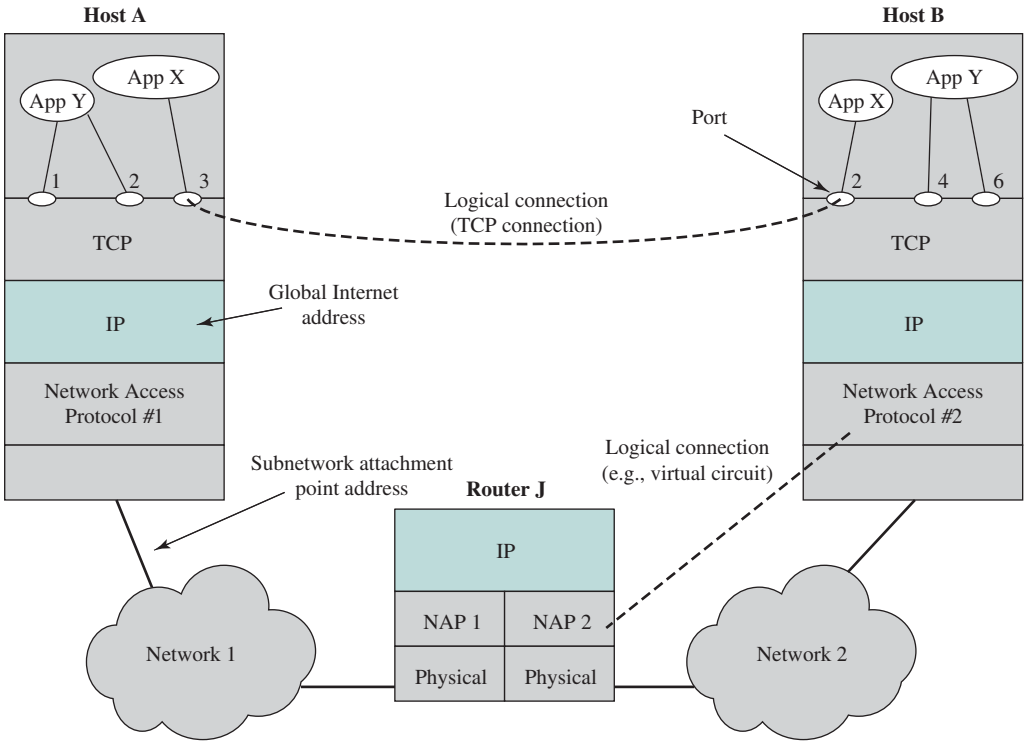


Figure 17.4 TCP/IP Concepts

the proper host. This address is used by IP for routing and delivery. Each application within a host must have an address that is unique within the host; this allows the host-to-host protocol (TCP) to deliver data to the proper process. These latter addresses are known as **ports**.

Let us trace a simple operation. Suppose a process, associated with port 3 at host A, wishes to send a message to another process, associated with port 2 at host B. The process at A hands the message down to TCP with instructions to send it to host B, port 2. TCP hands the message down to IP with instructions to send it to host B. Note IP need not be told the identity of the destination port. All it needs to know is that the data are intended for host B. Next, IP hands the message down to the network access layer (e.g., Ethernet logic) with instructions to send it to router J (the first hop on the way to B).

To control this operation, control information as well as user data must be transmitted, as suggested in Figure 17.5. Let us say that the sending process generates a block of data and passes this to TCP. TCP may break this block into smaller pieces to make it more manageable. To each of these pieces, TCP appends control information known as the TCP header (see Figure 17.2a), forming a **TCP segment**. The control information is to be used by the peer TCP protocol entity at host B.

Next, TCP hands each segment over to IP, with instructions to transmit it to B. These segments must be transmitted across one or more networks and relayed

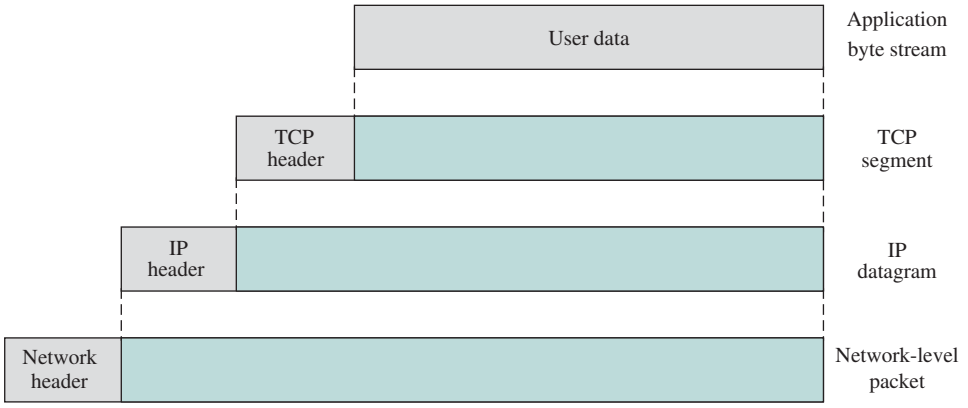


Figure 17.5 Protocol Data Units (PDUs) in the TCP/IP Architecture

through one or more intermediate routers. This operation, too, requires the use of control information. Thus IP appends a header of control information (see Figure 17.3) to each segment to form an **IP datagram**. An example of an item stored in the IP header is the destination host address (in this example, B).

Finally, each IP datagram is presented to the network access layer for transmission across the first network in its journey to the destination. The network access layer appends its own header, creating a packet, or frame. The packet is transmitted across the network to router J. The packet header contains the information that the network needs in order to transfer the data across the network. Examples of items that may be contained in this header include:

- **Destination network address:** The network must know to which attached device the packet is to be delivered, in this case router J.
- **Facilities requests:** The network access protocol might request the use of certain network facilities, such as priority.

At router J, the packet header is stripped off and the IP header examined. On the basis of the destination address information in the IP header, the IP module in the router directs the datagram across network 2 to B. To do this, the datagram is again augmented with a network access header.

When the data are received at B, the reverse process occurs. At each layer, the corresponding header is removed, and the remainder is passed on to the next higher layer, until the original user data are delivered to the destination process.

TCP/IP Applications

A number of applications have been standardized to operate on top of TCP. We mention three of the most common here.

The **Simple Mail Transfer Protocol (SMTP)** provides a basic electronic mail facility. It provides a mechanism for transferring messages among separate hosts. Features of SMTP include mailing lists, return receipts, and forwarding. The SMTP protocol does not specify the way in which messages are to be created; some local

editing or native electronic mail facility is required. Once a message is created, SMTP accepts the message and makes use of TCP to send it to an SMTP module on another host. The target SMTP module will make use of a local electronic mail package to store the incoming message in a user's mailbox.

The **File Transfer Protocol (FTP)** is used to send files from one system to another under user command. Both text and binary files are accommodated, and the protocol provides features for controlling user access. When a user wishes to engage in file transfer, FTP sets up a TCP connection to the target system for the exchange of control messages. This connection allows user ID and password to be transmitted and allows the user to specify the file and file actions desired. Once a file transfer is approved, a second TCP connection is set up for the data transfer. The file is transferred over the data connection, without the overhead of any headers or control information at the application level. When the transfer is complete, the control connection is used to signal the completion and to accept new file transfer commands.

SSH (Secure Shell) provides a secure remote logon capability, which enables a user at a terminal or personal computer to log on to a remote computer and function as if directly connected to that computer. SSH also supports file transfer between the local host and a remote server. SSH enables the user and the remote server to authenticate each other; it also encrypts all traffic in both directions. SSH traffic is carried on a TCP connection.

17.3 SOCKETS[1]

The concept of sockets and sockets programming was developed in the 1980s in the UNIX environment as the Berkeley Sockets Interface. In essence, a socket enables communication between a client and server process and may be either connection oriented or connectionless. A socket can be considered an endpoint in a communication. A client socket in one computer uses an address to call a server socket on another computer. Once the appropriate sockets are engaged, the two computers can exchange data.

Typically, computers with server sockets keep a TCP or UDP port open, ready for unscheduled incoming calls. The client typically determines the socket identification of the desired server by finding it in a Domain Name System (DNS) database. Once a connection is made, the server switches the dialogue to a different port number to free up the main port number for additional incoming calls.

Internet applications, such as TELNET and remote login (rlogin), make use of sockets, with the details hidden from the user. However, sockets can be constructed from within a program (in a language such as C or Java), enabling the programmer to easily support networking functions and applications. The sockets programming mechanism includes sufficient semantics to permit unrelated processes on different hosts to communicate.

The Berkeley Sockets Interface is the de facto standard **application programming interface (API)** for developing networking applications, spanning a wide range

[1]This section provides a Sockets overview. Appendix M contains a more detailed treatment.

of operating systems. Windows Sockets (WinSock) is based on the Berkeley specification. The sockets API provide generic access to interprocess communications services. Thus, the sockets capability is ideally suited for students to learn the principles of protocols and distributed applications by hands-on program development.

The Socket

Recall that each TCP and UDP header includes source port and destination port fields (see Figure 17.2). These port values identify the respective users (applications) of the two TCP entities. Also, each IPv4 and IPv6 header includes source address and destination address fields (see Figure 17.3); these **IP addresses** identify the respective host systems. The concatenation of a port value and an IP address forms a **socket**, which is unique throughout the Internet. Thus, in Figure 17.4, the combination of the IP address for host B and the port number for application X uniquely identifies the socket location of application X in host B. As the figure indicates, an application may have multiple socket addresses, one for each port into the application.

The socket is used to define an API, which is a generic communication interface for writing programs that use TCP or UDP. In practice, when used as an API, a socket is identified by the triple (protocol, local address, and local process). The local address is an IP address and the local process is a port number. Because port numbers are unique within a system, the port number implies the protocol (TCP or UDP). However, for clarity and ease of implementation, sockets used for an API include the protocol as well as the IP address and port number in defining a unique socket.

Corresponding to the two protocols, the Sockets API recognizes two types of sockets: stream sockets and datagram sockets. **Stream sockets** make use of TCP, which provides a connection-oriented reliable data transfer. Therefore, with stream sockets, all blocks of data sent between a pair of sockets are guaranteed for delivery and arrive in the order in which they were sent. **Datagram sockets** make use of UDP, which does not provide the connection-oriented features of TCP. Therefore, with datagram sockets, delivery is not guaranteed, nor is order necessarily preserved.

There is a third type of socket provided by the Sockets API: raw sockets. **Raw sockets** allow direct access to lower-layer protocols, such as IP.

Socket Interface Calls

This subsection summarizes the key system calls.

SOCKET SETUP The first step in using Sockets is to create a new socket using the `socket()` command. This command includes three parameters, the protocol family is always `PF_INET`, for the TCP/IP protocol suite. *Type* specifies whether this is a stream or datagram socket, and *protocol* specifies either TCP or UDP. The reason that both *type* and *protocol* need to be specified is to allow additional transport-level protocols to be included in a future implementation. Thus, there might be more than one datagram-style transport protocol, or more than one connection-oriented transport protocol. The `socket()` command returns an integer result that identifies this socket; it is similar to a UNIX file descriptor. The exact socket data structure depends on the implementation. It includes the source port and IP address and, if a

connection is open or pending, the destination port and IP address and various options and parameters associated with the connection.

After a socket is created, it must have an address to which to listen. The bind() function binds a socket to a socket address. The address has the structure

```
struct sockaddr_in {
    short int sin_family;         // Address family (TCP/IP)
    unsigned short int sin_port;// Port number
    struct in_addr sin_addr;     // Internet address
    unsigned char sin_zero[8];  // Same size as struct
};                                  sockaddr
```

SOCKET CONNECTION For a stream socket, once the socket is created, a connection must be set up to a remote socket. One side functions as a client and requests a connection to the other side, which acts as a server.

The server side of a connection setup requires two steps. First, a server application issues a `listen()`, indicating the given socket is ready to accept incoming connections. The parameter *backlog* is the number of connections allowed on the incoming queue. Each incoming connection is placed in this queue until a matching `accept()` is issued by the server side. Next, the `accept()` call is used to remove one request from the queue. If the queue is empty, the `accept()` blocks the process until a connection request arrives. If there is a waiting call, then `accept()` returns a new file descriptor for the connection. This creates a new socket, which has the IP address and port number of the remote party, the IP address of this system, and a new port number. The reason that a new socket with a new port number is assigned is that this enables the local application to continue to listen for more requests. As a result, an application may have multiple connections active at any time, each with a different local port number. This new port number is returned across the TCP connection to the requesting system.

A client application issues a `connect()` that specifies both a local socket and the address of a remote socket. If the connection attempt is unsuccessful `connect()` returns the value −1. If the attempt is successful, `connect()` returns a 0 and fills in the file descriptor parameter to include the IP address and port number of the local and foreign sockets. Recall that the remote port number may differ from that specified in the `foreignAddress` parameter because the port number is changed on the remote host.

Once a connection is set up, `getpeername()` can be used to find out who is on the other end of the connected stream socket. The function returns a value in the `sockfd` parameter.

SOCKET COMMUNICATION For **stream communication**, the functions `send()` and `recv()` are used to send or receive data over the connection identified by the sockfd parameter. In the `send()` call, the `*msg` parameter points to the block of data to be sent, and the `len` parameter specifies the number of bytes to be sent. The `flags` parameter contains control flags, typically set to 0. The `send()` call returns the number of bytes sent, which may be less than the number specified in the len parameter. In the `recv()` call, the `*buf` parameter points to the buffer for storing incoming data, with an upper limit on the number of bytes set by the `len` parameter.

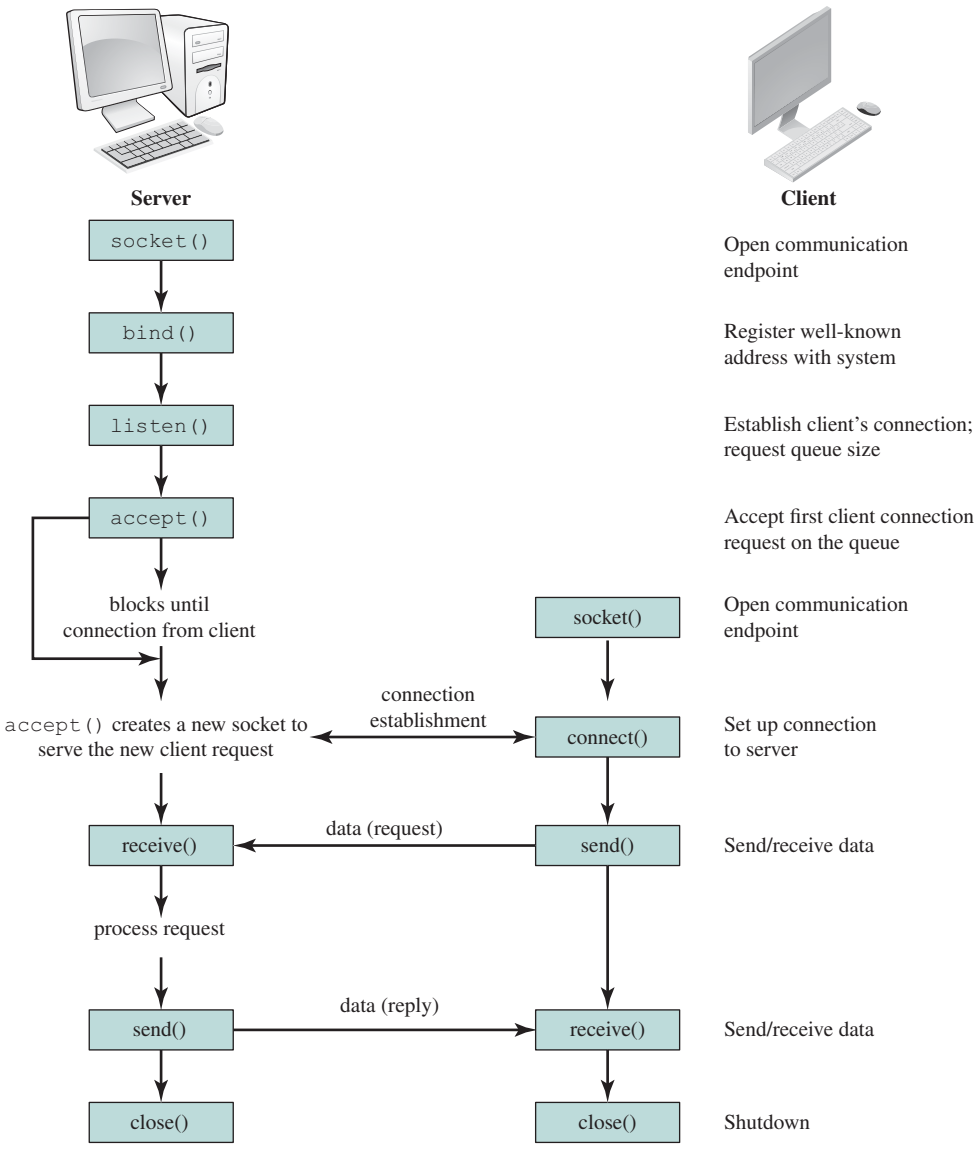


Figure 17.6 Socket System Calls for Connection-Oriented Protocol

At any time, either side can close the connection with the `close()` call, which prevents further sends and receives. The `shutdown()` call allows the caller to terminate sending or receiving or both.

Figure 17.6 shows the interaction of the clients and server sides in setting up, using, and terminating a connection.

For **datagram communication**, the functions `sendto()` and `recvfrom()` are used. The `sendto()` call includes all the parameters of the `send()` call plus a specification of the destination address (IP address and port). Similarly, the `recvfrom()` call includes an address parameter, which is filled in when data are received.

17.4 LINUX NETWORKING

Linux supports a variety of networking architectures, in particular TCP/IP by means of Berkeley Sockets. Figure 17.7 shows the overall structure of Linux support for TCP/IP. User-level processes interact with networking devices by means of system calls to the Sockets interface. The Sockets module in turn interacts with a software package in the kernel that handles transport-layer (TCP and UDP) and IP protocol operations. This software package exchanges data with the device driver for the network interface card.

Linux implements sockets as special files. Recall from Chapter 12 that, in UNIX systems, a special file is one that contains no data but provides a mechanism to map

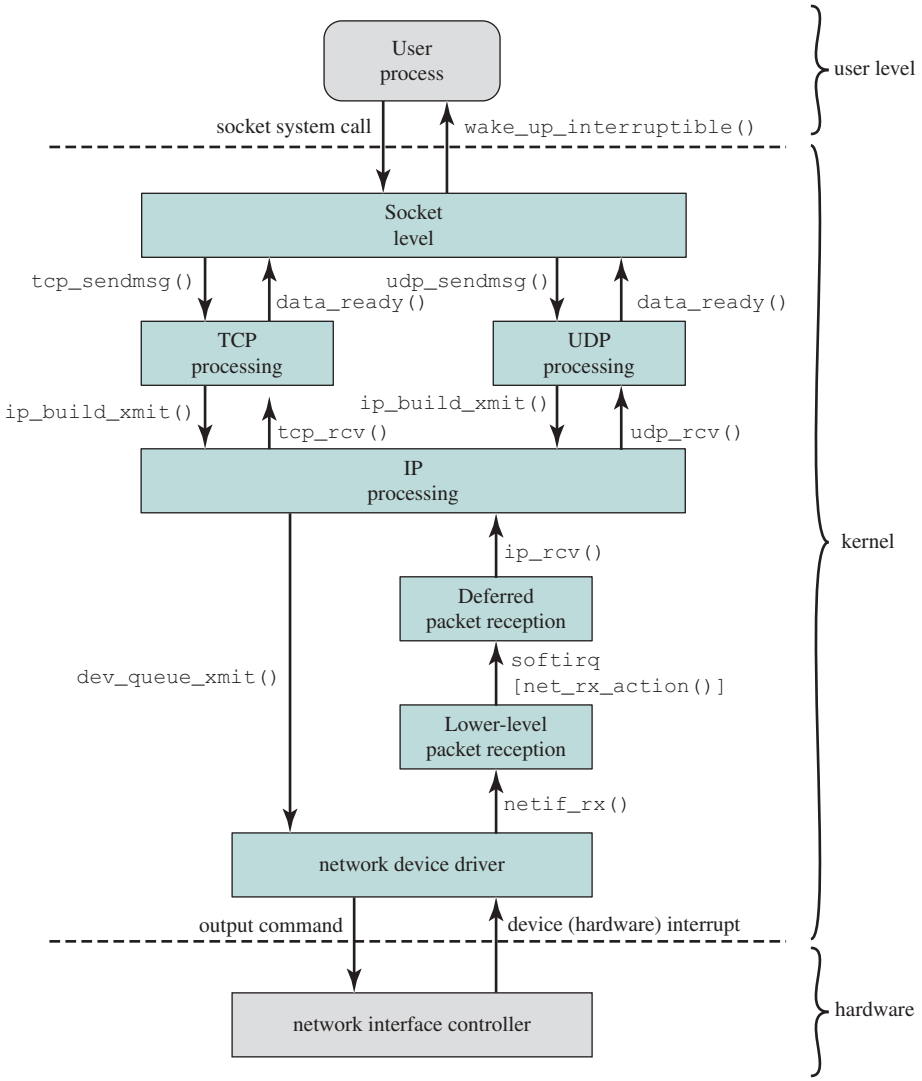


Figure 17.7 Linux Kernel Components for TCP/IP Processing

physical devices to file names. For every new socket, the Linux kernel creates a new inode in the *sockfs* special file system.

Figure 17.7 depicts the relationships among various kernel modules involved in sending and receiving TCP/IP-based data blocks. The remainder of this section looks at the sending and receiving facilities.

Sending Data

A user process uses the sockets calls described in Section 17.3 to create new sockets, set up connections to remote sockets, and send and receive data. To send data, the user process writes data to the socket with the following file system call:

```
write(sockfd, mesg, mesglen)
```

where `mesglen` is the length of the `mesg` buffer in bytes.

This call triggers the `write` method of the file object associated with the `sockfd` file descriptor. The file descriptor indicates whether this is a socket set up for TCP or UDP. The kernel allocates the appropriate data structures and invokes the appropriate sockets-level function to pass data to either a TCP module or a UDP module. The corresponding functions are `tcp_sendmsg()` and `udp_sendmsg()`, respectively. The transport-layer module allocates a data structure of the TCP or UPD header and performs `ip_build_xmit()` to invoke the IP-layer processing module. This module builds an IP datagram for transmission and places it in a transmission buffer for this socket. The IP-layer module then performs `dev_queue_xmit()` to queue the socket buffer for later transmission via the network device driver. When it is available, the network device driver will transmit buffered packets.

Receiving Data

Data reception is an unpredictable event and so involves the use of interrupts and deferrable functions. When an IP datagram arrives, the network interface controller issues a hardware interrupt to the corresponding network device driver. The interrupt triggers an interrupt service routine that handles the interrupt as part of the network device driver module. The driver allocates a kernel buffer for the incoming data block and transfers the data from the device controller to the buffer. The driver then performs `netif_rx()` to invoke a lower-level packet reception routine. In essence, the `netif_rx()` function places the incoming data block in a queue then issues a soft interrupt request (`softirq`) so the queued data will eventually be processed. The action to be performed when the `softirq` is processed is the `net_rx_action()` function.

Once a `softirq` has been queued, processing of this packet is halted until the kernel executes the `softirq` function, which is equivalent to saying until the kernel responds to this soft interrupt request and executes the function (in this case, `net_rx_action()`) associated with this soft interrupt. There are three places in the kernel, where the kernel checks to see if any `softirqs` are pending: when a hardware interrupt has been processed, when an application-level process invokes a system call, and when a new process is scheduled for execution.

When the `net_rx_action()` function is performed, it retrieves the queued packet and passes it on to the IP packet handler by means of an `ip_rcv` call. The IP packet handler processes the IP header then uses `tcp_rcv` or `udp_rcv` to invoke the transport-layer processing module. The transport-layer module processes the

transport-layer header and passes the data to the user through the sockets interface by means of a `wake_up_interruptible()` call, which awakens the receiving process.

17.5 SUMMARY

The communication functionality required for distributed applications is quite complex. This functionality is generally implemented as a structured set of modules. The modules are arranged in a vertical, layered fashion, with each layer providing a particular portion of the needed functionality and relying on the next lower layer for more primitive functions. Such a structure is referred to as a protocol architecture.

One motivation for the use of this type of structure is that it eases the task of design and implementation. It is standard practice for any large software package to break up the functions into modules that can be designed and implemented separately. After each module is designed and implemented, it can be tested. Then the modules can be combined and tested together. This motivation has led computer vendors to develop proprietary layered-protocol architectures. An example of this is the Systems Network Architecture (SNA) of IBM.

A layered architecture can also be used to construct a standardized set of communication protocols. In this case, the advantages of modular design remain. But, in addition, a layered architecture is particularly well-suited to the development of standards. Standards can be developed simultaneously for protocols at each layer of the architecture. This breaks down the work to make it more manageable and speeds up the standards-development process. The TCP/IP protocol architecture is the standard architecture used for this purpose. This architecture contains five layers. Each layer provides a portion of the total communications function required for distributed applications. Standards have been developed for each layer. Development work continues, particularly at the top (application) layer, where new distributed applications are still being defined.

17.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

application layer	protocol	syntax
application programming interface (API)	protocol architecture	network access layer
datagram communication	raw socket	TCP segment
datagram socket	router	TELNET
File Transfer Protocol (FTP)	semantics	timing
Internet Protocol (IP)	Simple Mail Transfer Protocol (SMTP)	Transmission Control Protocol (TCP)
IP addresses	socket	transport layer
IP datagram	SSH (secure shell)	User Datagram Protocol (UDP)
physical layer	stream communication	
port	stream socket	

Review Questions

17.1. What is the major function of the network access layer?
17.2. What tasks are performed by the transport layer?
17.3. What is a protocol?
17.4. What is a protocol architecture?
17.5. What is TCP/IP?
17.6. What is the purpose of the Sockets interface?

Problems

17.1. For this problem, first consider the case where you wish to order pizza for a party of guests. The layer models in Figure 17.8 can be used to describe the ordering and delivery of a pizza. The guest effectively places the order with the cook. The host communicates this order to the clerk, who places the order with the cook. The phone system provides the physical means for the order to be transported from host to clerk. The cook gives the pizza to the clerk with the order form (acting as a "header" to the pizza). The clerk boxes the pizza with the delivery address, and the delivery van encloses all of the orders to be delivered. The road provides the physical path for the delivery.

a. The French and Chinese prime ministers need to come to an agreement by telephone, but neither speaks the other's language. Further, neither has on hand a translator that can translate to the language of the other. However, both prime ministers have English translators on their staffs. Draw a diagram similar to Figure 17.8 to depict the situation, and describe the interaction at each layer.

b. Now suppose the Chinese prime minister's translator can translate only into Japanese and the French prime minister has a German translator available. A translator between German and Japanese is available in Germany. Draw a new diagram that reflects this arrangement, and describe the hypothetical phone conversation.

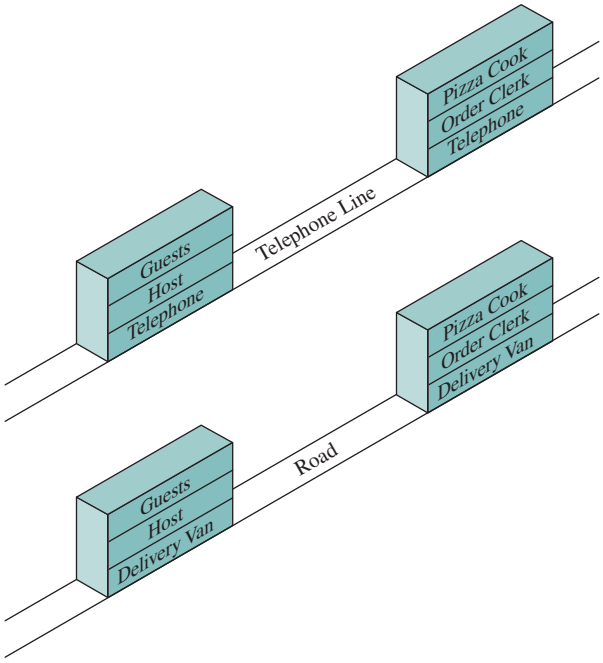


Figure 17.8 Architecture for Problem 17.1

17.2. List the major disadvantages of the layered approach to protocols.

17.3. A TCP segment consisting of 1,500 bits of data and 160 bits of header is sent to the IP layer, which appends another 160 bits of header. This is then transmitted through two networks, each of which uses a 24-bit packet header. The destination network has a maximum packet size of 800 bits. How many bits, including headers, are delivered to the network layer protocol at the destination?

17.4. Why does the TCP header have a header length field while the UDP header does not?

17.5. The previous version of the TFTP specification, RFC 783, included the following statement:

All packets other than those used for termination are acknowledged individually unless a timeout occurs.

The new specification revises this to say

All packets other than duplicate ACKs and those used for termination are acknowledged unless a timeout occurs.

The change was made to fix a problem referred to as the "Sorcerer's Apprentice." Deduce and explain the problem.

17.6. What is the limiting factor in the time required to transfer a file using TFTP?

17.7. A user on a UNIX host wants to transfer a 4,000-byte text file to a Microsoft Windows host. In order to do this, he transfers the file by means of TFTP, using the netascii transfer mode. Even though the transfer was reported as being performed successfully, the Windows host reports the resulting file size is 4,050 bytes, rather than the original 4,000 bytes. Does this difference in the file sizes imply an error in the data transfer? Why or why not?

17.8. The TFTP specification (RFC 1350) states that the transfer identifiers (TIDs) chosen for a connection should be randomly chosen, so the probability that the same number is chosen twice in immediate succession is very low. What would be the problem of using the same TIDs twice in immediate succession?

17.9. In to retransmit lost packets, TFTP must keep a copy of the data it sends. How many packets of data must TFTP keep at a time to implement this retransmission mechanism?

17.10. TFTP, like most protocols, will never send an error packet in response to an error packet it receives. Why?

17.11. We have seen that in order to deal with lost packets, TFTP implements a time-out-and-retransmit scheme, by setting a retransmission timer when it transmits a packet to the remote host. Most TFTP implementations set this timer to a fixed value of about five seconds. Discuss the advantages and the disadvantages of using a fixed value for the retransmission timer.

17.12. TFTP's time-out-and-retransmission scheme implies that all data packets will eventually be received by the destination host. Will these data also be received uncorrupted? Why or why not?

17.13. This chapter mentions the use of Frame Relay as a specific protocol or system used to connect to a wide area network. Each organization will have a certain collection of services available (like Frame Relay) but this is dependent upon provider provisioning, cost and customer premises equipment. What are some of the services available to you in your area?

17.14. Wireshark is a free packet sniffer that allows you to capture traffic on a local area network. It can be used on a variety of operating systems and is available at www.ethereal.com. You must also install the WinPcap packet capture driver, which can be obtained from www.wireshark.org/.

After starting a capture from Wireshark, start a TCP-based application like TELNET, FTP, or HTTP (Web browser). Can you determine the following from your capture?

a. Source and destination layer 2 addresses (MAC).

b. Source and destination layer 3 addresses (IP).

c. Source and destination layer 4 addresses (port numbers).

17.15. Packet capture software or sniffers can be powerful management and security tools. By using the filtering capability that is built in, you can trace traffic based on several different criteria and eliminate everything else. Use the filtering capability built into Ethereal to do the following;

a. Capture only traffic coming from your computer's MAC address.

b. Capture only traffic coming from your computer's IP address.

c. Capture only UDP-based transmissions.

APPENDIX 17A THE TRIVIAL FILE TRANSFER PROTOCOL

This appendix provides an overview of the Internet standard Trivial File Transfer Protocol (TFTP), defined in RFC 1350. Our purpose is to give the reader some flavor for the elements of a protocol. TFTP is simple enough to provide a concise example but includes most of the significant elements found in other, more complex, protocols.

Introduction to TFTP

TFTP is far simpler than the Internet standard File Transfer Protocol (FTP). There are no provisions for access control or user identification, so TFTP is only suitable for public access file directories. Because of its simplicity, TFTP is easily and compactly implemented. For example, some diskless devices use TFTP to download their firmware at boot time.

TFTP runs on top of UDP. The TFTP entity that initiates the transfer does so by sending a read or write request in a UDP segment with a destination port of 69 to the target system. This port is recognized by the target UDP module as the identifier of the TFTP module. For the duration of the transfer, each side uses a transfer identifier (TID) as its port number.

TFTP Packets

TFTP entities exchange commands, responses, and file data in the form of packets, each of which is carried in the body of a UDP segment. TFTP supports five types of packets (see Figure 17.9); the first two bytes contain an opcode that identifies the packet type:

- **RRQ:** The read request packet requests permission to transfer a file from the other system. The packet includes a file name, which is a sequence of ASCII[2] bytes terminated by a zero byte. The zero byte is the means by which the receiving TFTP entity knows when the file name is terminated. The packet also includes a mode field, which indicates whether the data file is to be interpreted as a string of ASCII bytes (netascii mode) or as raw 8-bit bytes (octet mode) of data. In netascii mode, the file is transferred as lines of characters, each terminated by a carriage return, line feed. Each system must translate between its own format for character files and the TFTP format.

[2]ASCII is the American Standard Code for Information Interchange, a standard of the American National Standards Institute. It designates a unique 7-bit pattern for each letter, with an eighth bit used for parity. ASCII is equivalent to the International Reference Alphabet (IRA), defined in ITU-T Recommendation T.50. See Appendix N for a discussion.

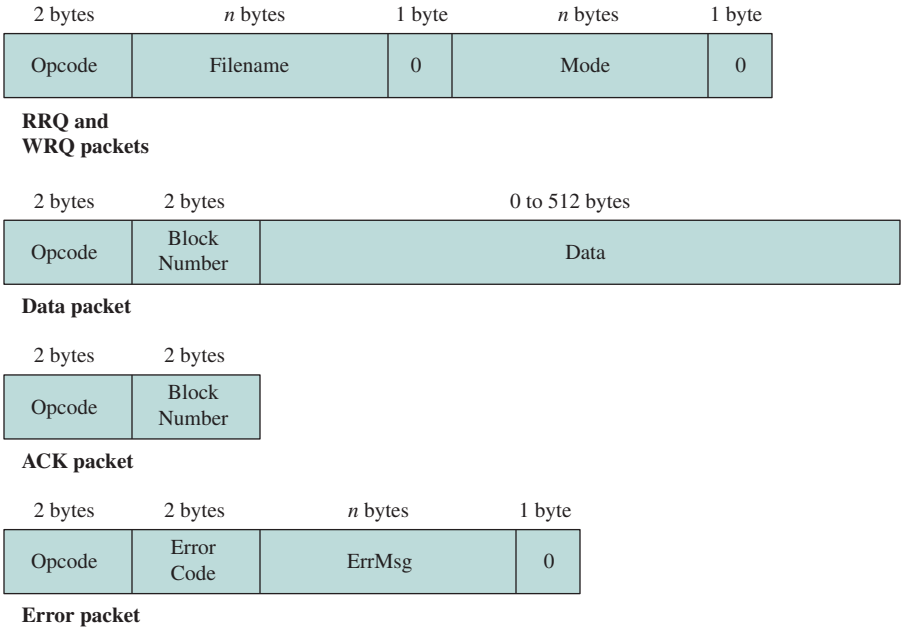


Figure 17.9 TFTP Packet Formats

- **WRQ:** The write request packet requests permission to transfer a file to the other system.
- **Data:** The block numbers on data packets begin with one and increase by one for each new block of data. This convention enables the program to use a single number to discriminate between new packets and duplicates. The data field is from 0 to 512 bytes long. If it is 512 bytes long, the block is not the last block of data; if it is from 0 to 511 bytes long, it signals the end of the transfer.
- **ACK:** This packet is used to acknowledge receipt of a data packet or a WRQ packet. An ACK of a data packet contains the block number of the data packet being acknowledged. An ACK of a WRQ contains a block number of zero.
- **Error:** An error packet can be the acknowledgment of any other type of packet. The error code is an integer indicating the nature of the error (see Table 17.1). The error message is intended for human consumption and should be in ASCII. Like all other strings, it is terminated with a zero byte.

All packets other than duplicate ACKs (explained subsequently) and those used for termination are to be acknowledged. Any packet can be acknowledged by an error packet. If there are no errors, then the following conventions apply. A WRQ or a data packet is acknowledged by an ACK packet. When an RRQ is sent, the other side responds (in the absence of error) by beginning to transfer the file; thus, the first data block serves as an acknowledgment of the RRQ packet. Unless a file transfer is complete, each ACK packet from one side is followed by a data packet from the other, so the data packet functions as an acknowledgment. An error packet can be acknowledged by any other kind of packet, depending on the circumstance.

Table 17.1 TFTP Error Codes

Value	Meaning
0	Not defined, see error message (if any)
1	File not found
2	Access violation
3	Disk full or allocation exceeded
4	Illegal TFTP operation
5	Unknown transfer ID
6	File already exists
7	No such user

Figure 17.10 shows a TFTP data packet in context. When such a packet is handed down to UDP, UDP adds a header to form a UDP segment. This is then passed to IP, which adds an IP header to form an IP datagram.

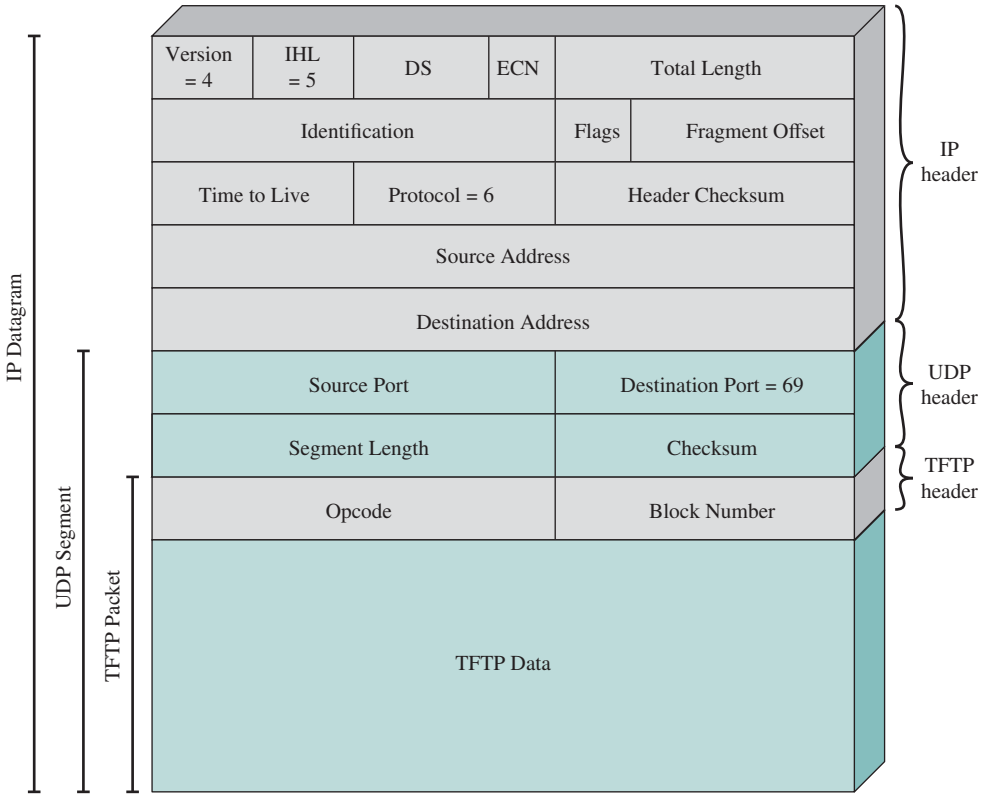


Figure 17.10 A TFTP Packet in Context

Overview of a Transfer

The example illustrated in Figure 17.11 is of a simple file transfer operation from A to B. No errors occur and the details of the option specification are not explored.

The operation begins when the TFTP module in system A sends a WRQ to the TFTP module in system B. The WRQ packet is carried as the body of a UDP segment. The WRQ includes the name of the file (in this case, XXX) and a mode of octet, or raw data. In the UDP header, the destination port number is 69, which alerts the receiving UDP entity that this message is intended for the TFTP application. The source port number is a TID selected by A, in this case 1511. System B is prepared to accept the file and so responds with an ACK with a block number of 0. In the UDP header, the destination port is 1511, which enables the UDP entity at A to route the incoming packet to the TFTP module, which can match this TID with

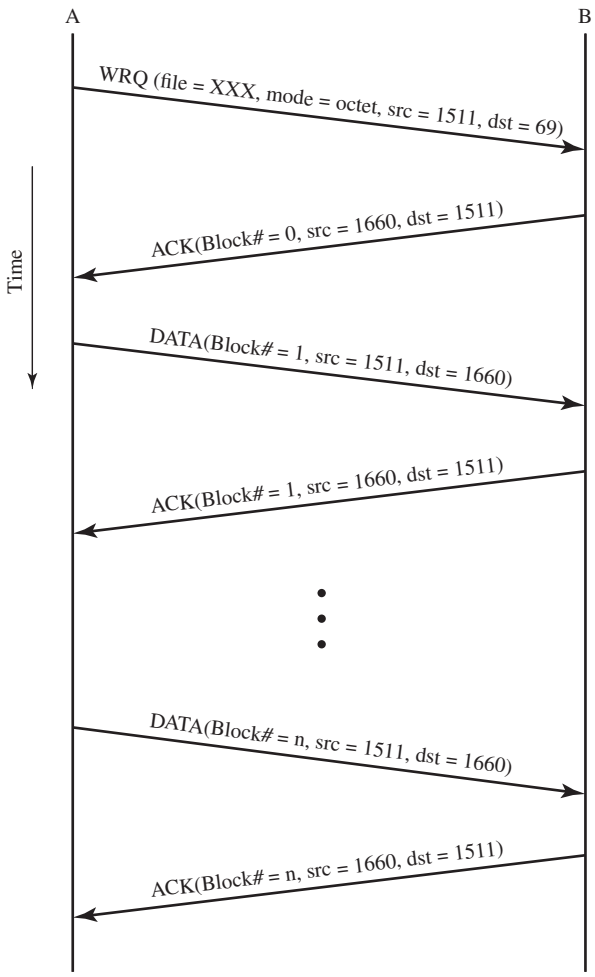


Figure 17.11 Example TFTP Operation

the TID in the WRQ. The source port is a TID selected by B for this file transfer, in this case 1660.

Following this initial exchange, the file transfer proceeds. The transfer consists of one or more data packets from A, each of which is acknowledged by B. The final data packet contains less than 512 bytes of data, which signals the end of the transfer.

Errors and Delays

If TFTP operates over a network or the Internet (as opposed to a direct data link), it is possible for packets to be lost. Because TFTP operates over UDP, which does not provide a reliable delivery service, there needs to be some mechanism in TFTP to deal with lost packets. TFTP uses the common technique of a time-out mechanism. Suppose A sends a packet to B that requires an acknowledgment (i.e., any packet other than duplicate ACKs and those used for termination). When A has transmitted the packet, it starts a timer. If the timer expires before the acknowledgment is received from B, A retransmits the same packet. If in fact the original packet was lost, then the retransmission will be the first copy of this packet received by B. If the original packet was not lost but the acknowledgment from B was lost, then B will receive two copies of the same packet from A and simply acknowledges both copies. Because of the use of block numbers, this causes no confusion. The only exception to this rule is for duplicate ACK packets. The second ACK is ignored.

Syntax, Semantics, and Timing

In Section 17.1, it was mentioned that the key features of a protocol can be classified as syntax, semantics, and timing. These categories are easily seen in TFTP. The formats of the various TFTP packets determine the **syntax** of the protocol. The **semantics** of the protocol are shown in the definitions of each of the packet types and the error codes. Finally, the sequence in which packets are exchanged, the use of block numbers, and the use of timers are all aspects of the **timing** of TFTP.

CHAPTER 18

Distributed Processing, Client/Server, and Clusters

Learning Objectives

After studying this chapter, you should be able to:

- Present a summary of the key aspects of client/server computing.
- Understand the principle design issues for distributed message passing.
- Understand the principle design issues for remote procedure calls.
- Understand the principle design issues for clusters.
- Describe the cluster mechanisms in Windows 7 and Beowulf.

In this chapter, we begin with an examination of some of the key concepts in distributed software, including client/server architecture, message passing, and remote procedure calls. Then we examine the increasingly important cluster architecture.

Chapters 17 and 18 complete our discussion of distributed systems.

18.1 CLIENT/SERVER COMPUTING

The concept of client/server computing, and related concepts, has become increasingly important in information technology systems. This section begins with a description of the general nature of client/server computing. This is followed by a discussion of alternative ways of organizing the client/server functions. The issue of **file cache consistency**, raised by the use of file servers, is then examined. Finally, this section introduces the concept of middleware.

What Is Client/Server Computing?

As with other new waves in the computer field, client/server computing comes with its own set of jargon words. Table 18.1 lists some of the terms that are commonly found in descriptions of client/server products and applications.

Table 18.1 Client/Server Terminology

Applications Programming Interface (API) A set of function and call programs that allow clients and servers to intercommunicate.
Client A networked information requester, usually a PC or workstation, that can query a database and/or other information from a server.
Middleware A set of drivers, APIs, or other software that improves connectivity between a client application and a server.
Relational Database A database in which information access is limited to the selection of rows that satisfy all search criteria.
Server A computer, usually a high-powered workstation, a minicomputer, or a mainframe, that houses information for manipulation by networked clients.
Structured Query Language (SQL) A language developed by IBM and standardized by ANSI for addressing, creating, updating, or querying relational databases.

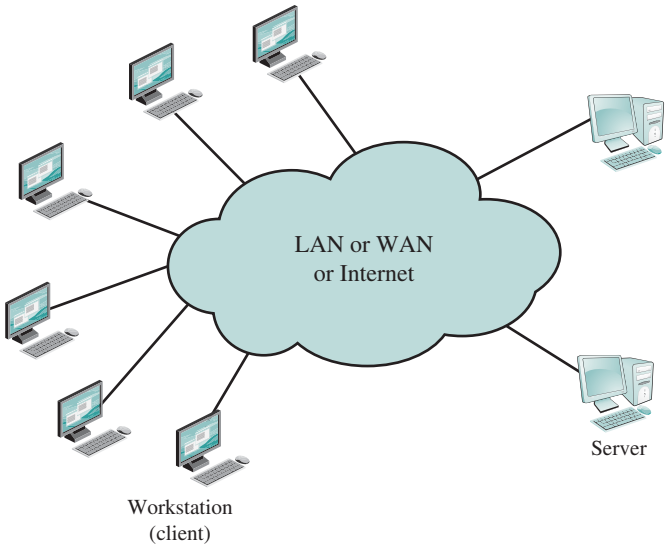


Figure 18.1 Generic Client/Server Environment

Figure 18.1 attempts to capture the essence of the client/server concept. As the term suggests, a *client/server environment* is populated by clients and servers. The **client** machines are generally single-user PCs or workstations that provide a user-friendly interface to the end user. The client-based station generally presents the type of graphical interface that is most comfortable to users, including the use of windows and a mouse. Microsoft Windows and Macintosh OS provide examples of such interfaces. Client-based applications are tailored for ease of use and include such familiar tools as the spreadsheet.

Each **server** in the client/server environment provides a set of shared services to the clients. The most common type of server currently is the database server, usually controlling a relational database. The server enables many clients to share access to the same database and enables the use of a high-performance computer system to manage the database.

In addition to clients and servers, the third essential ingredient of the client/server environment is the **network**. Client/server computing is typically distributed computing. Users, applications, and resources are distributed in response to business requirements and linked by a single LAN or WAN or by an internet of networks.

How does a client/server configuration differ from any other distributed processing solution? There are a number of characteristics that stand out and together, make client/server distinct from other types of distributed processing:

- There is a heavy reliance on bringing user-friendly applications to the user on his or her system. This gives the user a great deal of control over the timing and style of computer usage, and gives department-level managers the ability to be responsive to their local needs.
- Although applications are dispersed, there is an emphasis on centralizing corporate databases and many network management and utility functions. This

enables corporate management to maintain overall control of the total capital investment in computing and information systems and to provide interoperability so systems are tied together. At the same time, it relieves individual departments and divisions of much of the overhead of maintaining sophisticated computer-based facilities, but enables them to choose just about any type of machine and interface they need to access data and information.

- There is a commitment, both by user organizations and vendors, to open and modular systems. This means that the user has more choice in selecting products and in mixing equipment from a number of vendors.
- Networking is fundamental to the operation. Thus, network management and network security have a high priority in organizing and operating information systems.

Client/Server Applications

The key feature of a client/server architecture is the allocation of application-level tasks between clients and servers. Figure 18.2 illustrates the general case. In both client and server, of course, the basic software is an operating system running on the hardware platform. The platforms and the operating systems of client and server may differ. Indeed, there may be a number of different types of client platforms and operating systems and a number of different types of server platforms in a single environment. As long as a particular client and server share the same communications protocols and support the same applications, these lower-level differences are irrelevant.

It is the communications software that enables client and server to interoperate. The principal example of such software is TCP/IP. Of course, the point of all of this support software (communications and operating system) is to provide a base for distributed applications. Ideally, the actual functions performed by the application can be split up between client and server in a way that optimizes the use of resources. In some cases, depending on the application needs, the bulk of the applications

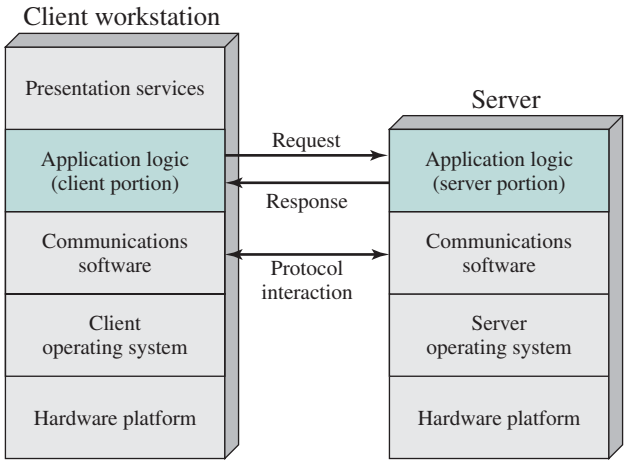


Figure 18.2 Generic Client/Server Architecture

software executes at the server, while in other cases, most of the application logic is located at the client.

An essential factor in the success of a client/server environment is the way in which the user interacts with the system as a whole. Thus, the design of the user interface on the client machine is critical. In most client/server systems, there is heavy emphasis on providing a **graphical user interface (GUI)** that is easy to use, easy to learn, yet powerful and flexible. Thus, we can think of a presentation services module in the client workstation that is responsible for providing a user-friendly interface to the distributed applications available in the environment.

DATABASE APPLICATIONS As an example that illustrates the concept of splitting application logic between client and server, let us consider one of the most common families of client/server applications: those that use relational databases. In this environment, the server is essentially a database server. Interaction between client and server is in the form of transactions in which the client makes a database request and receives a database response.

Figure 18.3 illustrates, in general terms, the architecture of such a system. The server is responsible for maintaining the database, for which purpose a complex database management system software module is required. A variety of different applications that make use of the database can be housed on client machines. The "glue" that ties client and server together is software that enables the client to make requests for access to the server's database. A popular example of such logic is the structured query language (SQL).

Figure 18.3 suggests that all of the application logic—the software for "number crunching" or other types of data analysis—is on the client side, while the server is only concerned with managing the database. Whether such a configuration is appropriate depends on the style and intent of the application. For example, suppose the primary

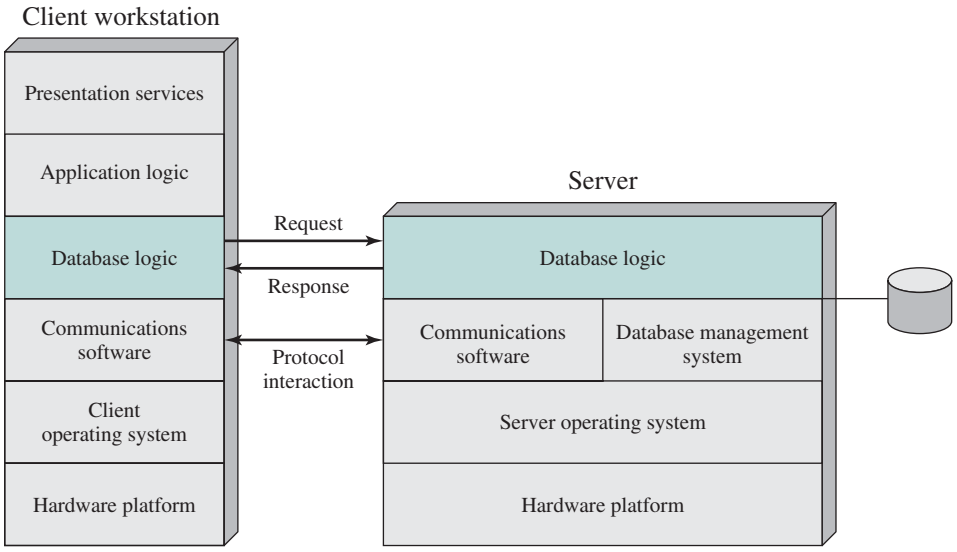
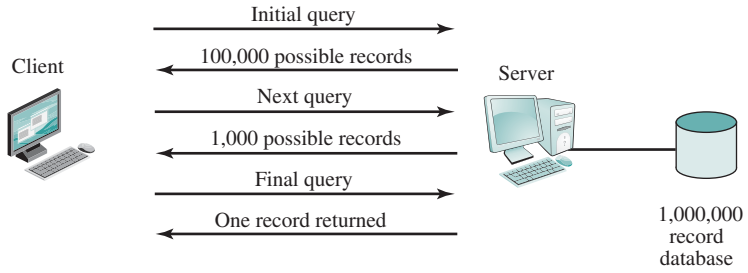


Figure 18.3 Client/Server Architecture for Database Applications

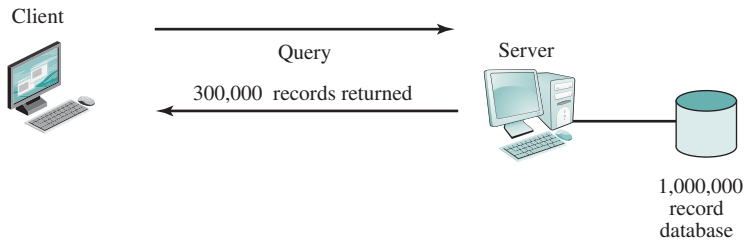
purpose is to provide online access for record lookup. Figure 18.4a suggests how this might work. Suppose the server is maintaining a database of 1 million records (called rows in relational database terminology), and the user wants to perform a lookup that should result in zero, one, or at most a few records. The user could search for these records using a number of search criteria (e.g., records older than 1992, records referring to individuals in Ohio, records referring to a specific event or characteristic, etc.). An initial client query may yield a server response that there are 100,000 records that satisfy the search criteria. The user then adds additional qualifiers and issues a new query. This time, a response indicating that there are 1,000 possible records is returned. Finally, the client issues a third request with additional qualifiers. The resulting search criteria yield a single match, and the record is returned to the client.

The preceding application is well-suited to a client/server architecture for two reasons:

1. There is a massive job of sorting and searching the database. This requires a large disk or bank of disks, a high-speed CPU, and a high-speed I/O architecture. Such capacity and power is not needed and is too expensive for a single-user workstation or PC.
2. It would place too great a traffic burden on the network to move the entire 1-million-record file to the client for searching. Therefore, it is not enough for the server just to be able to retrieve records on behalf of a client; the server needs to have database logic that enables it to perform searches on behalf of a client.



(a) Desirable client/server use



(b) Misused client/server

Figure 18.4 Client/Server Database Usage

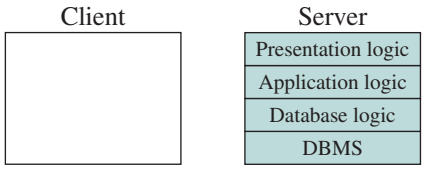
Now consider the scenario of Figure 18.4b, which has the same 1-million-record database. In this case, a single query results in the transmission of 300,000 records over the network. This might happen if, for example, the user wishes to find the grand total or mean value of some field across many records or even the entire database.

Clearly, this latter scenario is unacceptable. One solution to this problem, which maintains the client/server architecture with all of its benefits, is to move part of the application logic over to the server. That is, the server can be equipped with application logic for performing data analysis as well as data retrieval and data searching.

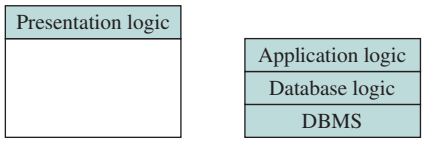
CLASSES OF CLIENT/SERVER APPLICATIONS Within the general framework of client/server, there is a spectrum of implementations that divide the work between client and server differently. Figure 18.5 illustrates in general terms some of the major options for database applications. Other splits are possible, and the options may have a different characterization for other types of applications. In any case, it is useful to examine this figure to get a feel for the kind of trade-offs possible.

Figure 18.5 depicts four classes:

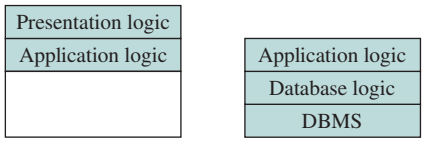
- **Host-based processing:** *Host-based processing* is not true client/server computing as the term is generally used. Rather, host-based processing refers to the



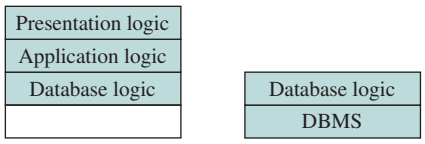
(a) Host-based processing



(b) Server-based processing



(c) Cooperative processing



(d) Client-based processing

Figure 18.5 Classes of Client/Server Applications

traditional mainframe environment in which all or virtually all of the processing is done on a central host. Often the user interface is via a dumb terminal. Even if the user is employing a microcomputer, the user's station is generally limited to the role of a terminal emulator.

- **Server-based processing:** The most basic class of client/server configuration is one in which the client is principally responsible for providing a graphical user interface, while virtually all of the processing is done on the server. This configuration is typical of early client/server efforts, especially departmental-level systems. The rationale behind such configurations is that the user workstation is best suited to providing a user-friendly interface, and that databases and applications can easily be maintained on central systems. Although the user gains the advantage of a better interface, this type of configuration does not generally lend itself to any significant gains in productivity, or to any fundamental changes in the actual business functions that the system supports.
- **Client-based processing:** At the other extreme, virtually all application processing may be done at the client, with the exception of data validation routines and other database logic functions that are best performed at the server. Generally, some of the more sophisticated database logic functions are housed on the client side. This architecture is perhaps the most common client/server approach in current use. It enables the user to employ applications tailored to local needs.
- **Cooperative processing:** In a cooperative processing configuration, the application processing is performed in an optimized fashion, taking advantage of the strengths of both client and server machines and of the distribution of data. Such a configuration is more complex to set up and maintain but, in the long run, this type of configuration may offer greater user productivity gains and greater network efficiency than other client/server approaches.

Figures 18.5c and 18.5d correspond to configurations in which a considerable fraction of the load is on the client. This so-called **fat client** model has been popularized by application development tools such as Sybase Inc.'s PowerBuilder and Gupta Corp.'s SQL Windows. Applications developed with these tools are typically departmental in scope. The main benefit of the fat client model is that it takes advantage of desktop power, offloading application processing from servers and making them more efficient and less likely to be bottlenecks.

There are, however, several disadvantages to the fat client strategy. The addition of more functions rapidly overloads the capacity of desktop machines, forcing companies to upgrade. If the model extends beyond the department to incorporate many users, the company must install high-capacity LANs to support the large volumes of transmission between the thin servers and the fat clients. Finally, it is difficult to maintain, upgrade, or replace applications distributed across tens or hundreds of desktops.

Figure 18.5b is representative of a **thin client** approach. This approach more nearly mimics the traditional host-centered approach and is often the migration path for evolving corporate-wide applications from the mainframe to a distributed environment.

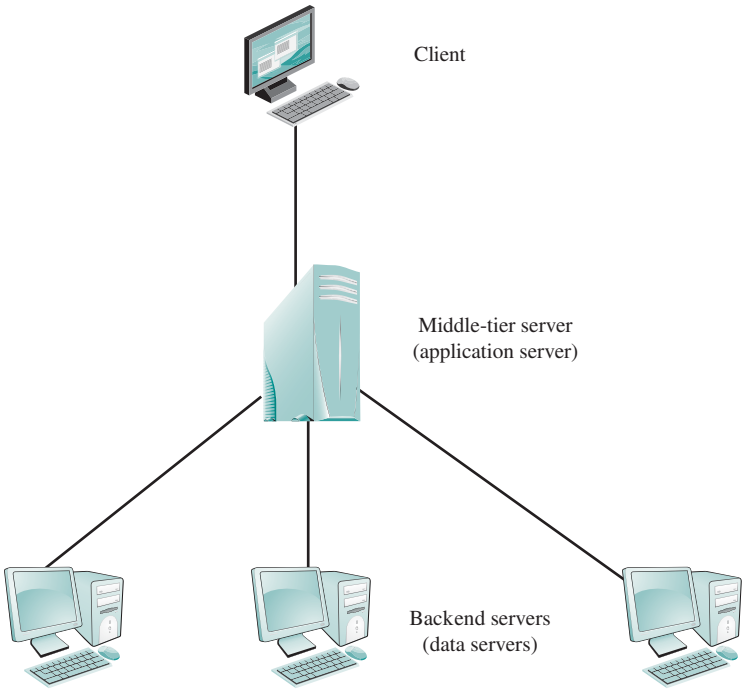


Figure 18.6 Three-Tier Client/Server Architecture

THREE-TIER CLIENT/SERVER ARCHITECTURE The traditional client/server architecture involves two levels, or tiers: a client tier and a server tier. A three-tier architecture is also common (see Figure 18.6). In this architecture, the application software is distributed among three types of machines: a user machine, a middle-tier server, and a backend server. The user machine is the client machine we have been discussing and, in the three-tier model, is typically a thin client. The middle-tier machines are essentially gateways between the thin user clients and a variety of backend database servers. The middle-tier machines can convert protocols and map from one type of database query to another. In addition, the middle-tier machine can merge/integrate results from different data sources. Finally, the middle-tier machine can serve as a gateway between the desktop applications and the backend legacy applications by mediating between the two worlds.

The interaction between the middle-tier server and the backend server also follows the client/server model. Thus, the middle-tier system acts as both a client and a server.

FILE CACHE CONSISTENCY When a file server is used, performance of file I/O can be noticeably degraded relative to local file access because of the delays imposed by the network. To reduce this performance penalty, individual systems can use file caches to hold recently accessed file records. Because of the principle of locality, use of a local file cache should reduce the number of remote server accesses that must be made.

Figure 18.7 illustrates a typical distributed mechanism for caching files among a networked collection of workstations. When a process makes a file access, the request is presented first to the cache of the process's workstation ("file traffic"). If not satisfied there, the request is passed either to the local disk, if the file is stored there ("disk traffic"), or to a file server, where the file is stored ("server traffic"). At the server, the server's cache is first interrogated and, if there is a miss, then the server's disk is accessed. The dual caching approach is used to reduce communications traffic (client cache) and disk I/O (server cache).

When caches always contain exact copies of remote data, we say the caches are **consistent**. It is possible for caches to become inconsistent when the remote data are changed and the corresponding obsolete local cache copies are not discarded. This can happen if one client modifies a file that is also cached by other clients. The difficulty is actually at two levels. If a client adopts a policy of immediately writing any changes to a file back to the server, then any other client that has a cache copy of the relevant portion of the file will have obsolete data. The problem is made even worse if the client delays writing back changes to the server. In that case, the server itself has an obsolete version of the file, and new file read requests to the server might obtain obsolete data. The problem of keeping local cache copies up to date to changes in remote data is known as the **cache consistency** problem.

The simplest approach to cache consistency is to use file-locking techniques to prevent simultaneous access to a file by more than one client. This guarantees consistency at the expense of performance and flexibility. A more powerful approach is provided with the facility in Sprite [NELS88, OUST88]. Any number of remote processes may open a file for read and create their own client cache. But when an open file request to a server requests write access and other processes have the file open for read access, the server takes two actions. First, it notifies the writing process that, although it may maintain a cache, it must write back all altered blocks immediately

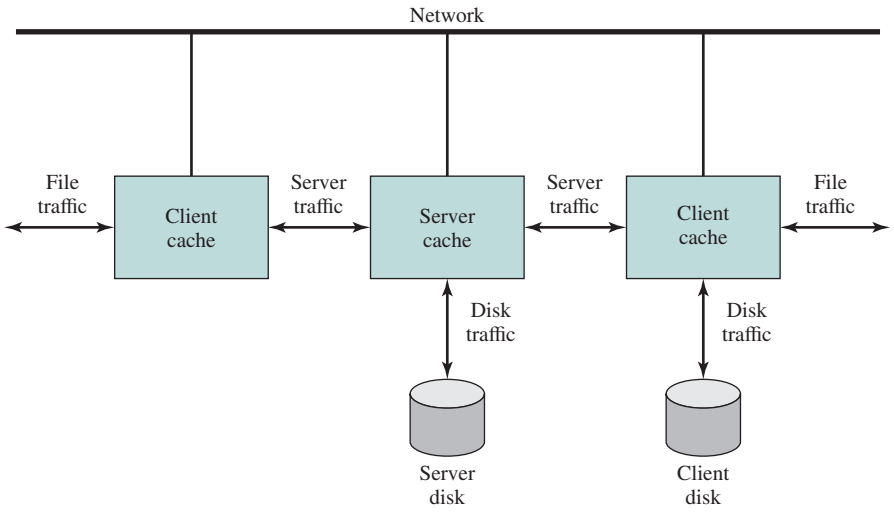


Figure 18.7 Distributed File Caching in Sprite

upon update. There can be at most one such client. Second, the server notifies all reading processes that have the file open that the file is no longer cacheable.

Middleware

The development and deployment of client/server products has far outstripped efforts to standardize all aspects of distributed computing, from the physical layer up to the application layer. This lack of standards makes it difficult to implement an integrated, multivendor, enterprise-wide client/server configuration. Because much of the benefit of the client/server approach is tied up with its modularity and the ability to mix and match platforms and applications to provide a business solution, this interoperability problem must be solved.

To achieve the true benefits of the client/server approach, developers must have a set of tools that provide a uniform means and style of access to system resources across all platforms. This will enable programmers to build applications that not only look and feel the same on various PCs and workstations, but that use the same method to access data regardless of the location of that data.

The most common way to meet this requirement is by the use of standard programming interfaces and protocols that sit between the application above and communications software and operating system below. Such standardized interfaces and protocols have come to be referred to as middleware. With standard programming interfaces, it is easy to implement the same application on a variety of server types and workstation types. This obviously benefits the customer, but vendors are also motivated to provide such interfaces. The reason is that customers buy applications, not servers; customers will only choose among those server products that run the applications they want. The standardized protocols are needed to link these various server interfaces back to the clients that need access to them.

There is a variety of middleware packages ranging from the very simple to the very complex. What they all have in common is the capability to hide the complexities and disparities of different network protocols and operating systems. Client and server vendors generally provide a number of the more popular middleware packages as options. Thus, a user can settle on a particular middleware strategy then assemble equipment from various vendors that support that strategy.

Middleware Architecture Figure 18.8 suggests the role of middleware in a client/server architecture. The exact role of the middleware component will depend on the style of client/server computing being used. Referring back to Figure 18.5, recall that there are a number of different client/server approaches, depending on the way in which application functions are split up. In any case, Figure 18.8 gives a good general idea of the architecture involved.

Note there is both a client and server component of middleware. The basic purpose of middleware is to enable an application or a user at a client to access a variety of services on servers without being concerned about differences among servers. To look at one specific application area, the structured query language (SQL) is supposed to provide a standardized means for access to a relational database by either a local or remote user or application. However, many relational database vendors, although they support SQL, have added their own proprietary extensions to

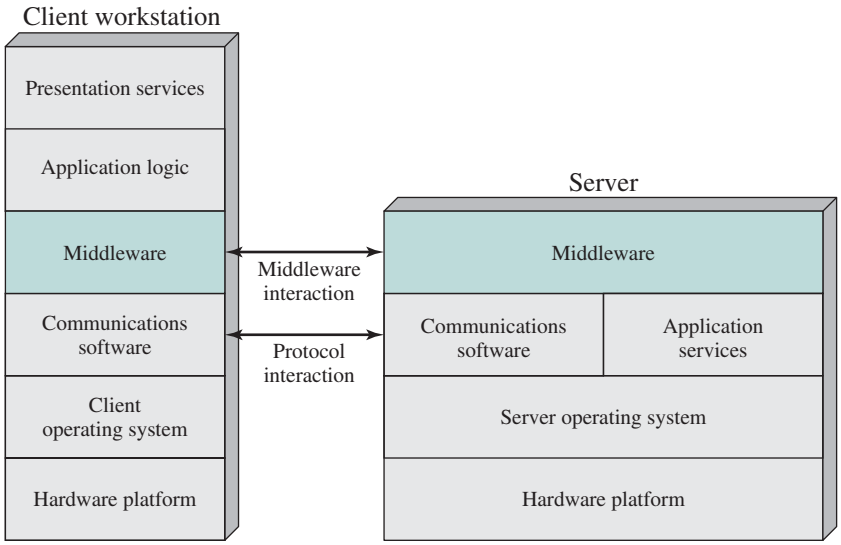


Figure 18.8 The Role of Middleware in Client/Server Architecture

SQL. This enables vendors to differentiate their products but also creates potential incompatibilities.

As an example, consider a distributed system used to support, among other things, the personnel department. The basic employee data, such as employee name and address, might be stored on a Gupta database, whereas salary information might be contained on an Oracle database. When a user in the personnel department requires access to particular records, that user does not want to be concerned with which vendor's database contains the records needed. Middleware provides a layer of software that enables uniform access to these differing systems.

It is instructive to look at the role of middleware from a logical, rather than an implementation, point of view. This viewpoint is illustrated in Figure 18.9. Middleware enables the realization of the promise of distributed client/server computing. The entire distributed system can be viewed as a set of applications and resources available to users. Users need not be concerned with the location of data or indeed the location of applications. All applications operate over a uniform applications programming interface (API). The middleware, which cuts across all client and server platforms, is responsible for routing client requests to the appropriate server.

Although there is a wide variety of middleware products, these products are typically based on one of two underlying mechanisms: message passing or remote procedure calls. These two methods are examined in the next two sections.

18.2 DISTRIBUTED MESSAGE PASSING

It is usually the case in a distributed processing systems that the computers do not share main memory; each is an isolated computer system. Thus, interprocessor communication techniques that rely on shared memory, such as semaphores, cannot be

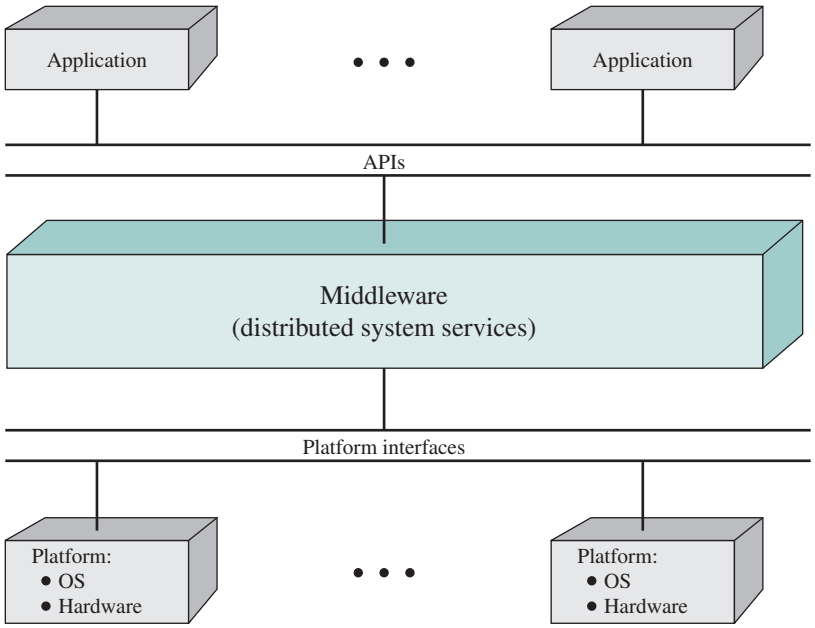


Figure 18.9 Logical View of Middleware

used. Instead, techniques that rely on message passing are used. In this section and the next, we look at the two most common approaches. The first is the straightforward application of messages as they are used in a single system. The second is a separate technique that relies on message passing as a basic function: the remote procedure call.

Figure 18.10a shows the use of message passing to implement client/server functionality. A client process requires some service (e.g., read a file, print) and sends a message containing a request for service to a server process. The server process honors the request and sends a message containing a reply. In its simplest form, only two functions are needed: Send and Receive. The Send function specifies a destination and includes the message content. The Receive function tells from whom a message is desired (including "all") and provides a buffer where the incoming message is to be stored.

Figure 18.11 suggests an implementation for message passing. Processes make use of the services of a message-passing module. Service requests can be expressed in terms of primitives and parameters. A primitive specifies the function to be performed, and the parameters are used to pass data and control information. The actual form of a primitive depends on the message-passing software. It may be a procedure call, or it may itself be a message to a process that is part of the operating system.

The Send primitive is used by the process that desires to send the message. Its parameters are the identifier of the destination process and the contents of the message. The message-passing module constructs a data unit that includes these two elements. This data unit is sent to the machine that hosts the destination process, using some sort of communications facility, such as TCP/IP. When the data unit is received in the target system, it is routed by the communications facility to the message-passing module. This module examines the process ID field and stores the message in the buffer for that process.

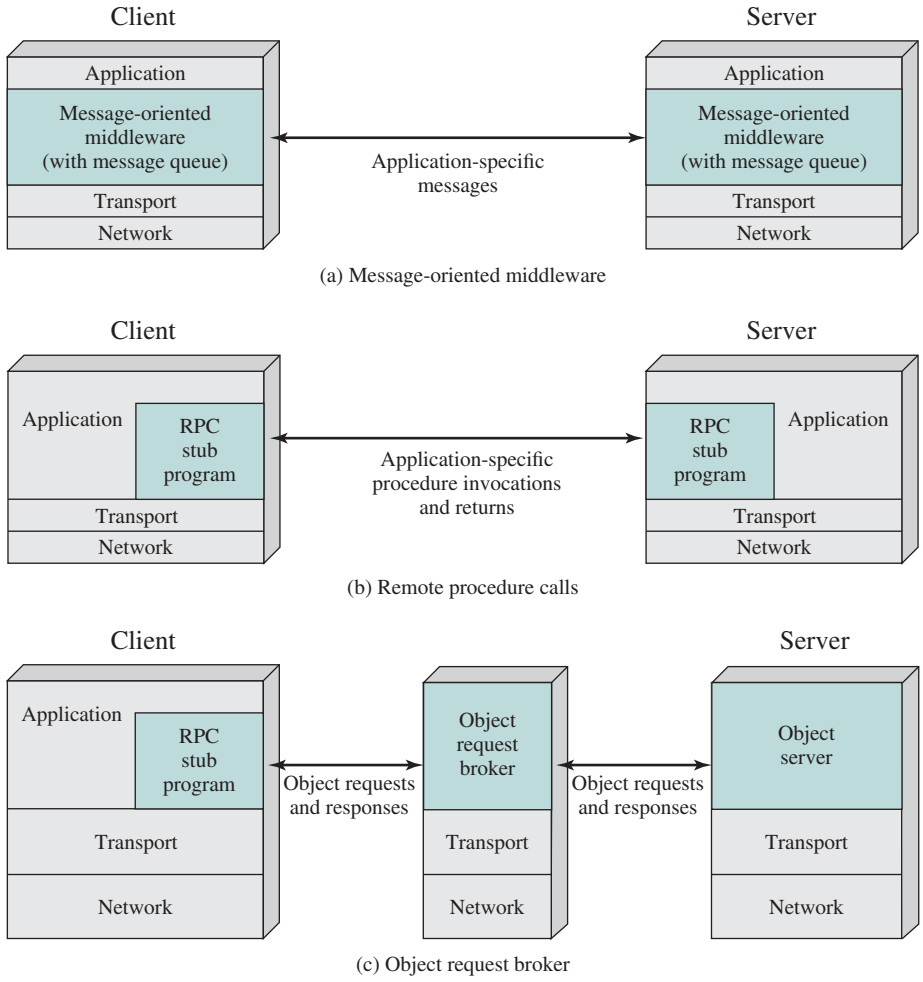


Figure 18.10 Middleware Mechanisms

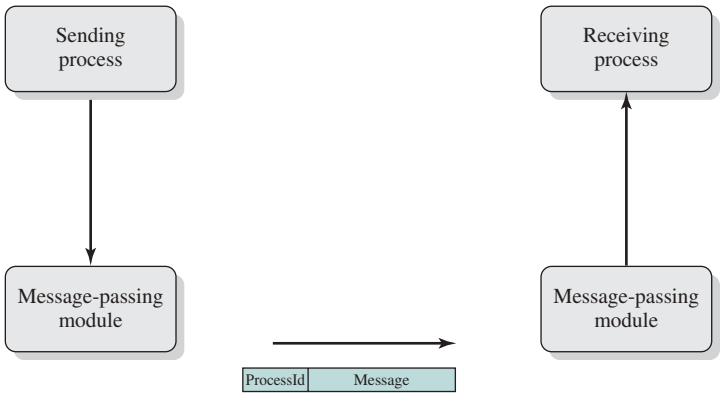


Figure 18.11 Basic Message-Passing Primitives

In this scenario, the receiving process must announce its willingness to receive messages by designating a buffer area and informing the message-passing module by a Receive primitive. An alternative approach does not require such an announcement. Instead, when the message-passing module receives a message, it signals the destination process with some sort of Receive signal then makes the received message available in a shared buffer.

Several design issues are associated with distributed message passing, and these are addressed in the remainder of this section.

Reliability versus Unreliability

A reliable message-passing facility is one that guarantees delivery if possible. Such a facility makes use of a reliable transport protocol or similar logic and performs error checking, acknowledgment, retransmission, and reordering of misordered messages. Because delivery is guaranteed, it is not necessary to let the sending process know the message was delivered. However, it might be useful to provide an acknowledgment back to the sending process so it knows that delivery has already taken place. In either case, if the facility fails to achieve delivery (e.g., persistent network failure, crash of destination system), the sending process is notified of the failure.

At the other extreme, the message-passing facility may simply send the message out into the communications network but will report neither success nor failure. This alternative greatly reduces the complexity and processing and communications overhead of the message-passing facility. For those applications that require confirmation that a message has been delivered, the applications themselves may use request and reply messages to satisfy the requirement.

Blocking versus Nonblocking

With nonblocking, or asynchronous, primitives, a process is not suspended as a result of issuing a Send or Receive. Thus, when a process issues a Send primitive, the operating system returns control to the process as soon as the message has been queued for transmission or a copy has been made. If no copy is made, any changes made to the message by the sending process before or even while it is being transmitted are made at the risk of the process. When the message has been transmitted or copied to a safe place for subsequent transmission, the sending process is interrupted to be informed that the message buffer may be reused. Similarly, a nonblocking Receive is issued by a process that then proceeds to run. When a message arrives, the process is informed by interrupt, or it can poll for status periodically.

Nonblocking primitives provide for efficient, flexible use of the message-passing facility by processes. The disadvantage of this approach is that it is difficult to test and debug programs that use these primitives. Irreproducible, timing-dependent sequences can create subtle and difficult problems.

The alternative is to use blocking, or synchronous, primitives. A blocking Send does not return control to the sending process until the message has been transmitted (unreliable service) or until the message has been sent and an acknowledgment received (reliable service). A blocking Receive does not return control until a message has been placed in the allocated buffer.

18.3 REMOTE PROCEDURE CALLS

A variation on the basic message-passing model is the remote procedure call. This is now a widely accepted and common method for encapsulating communication in a distributed system. The essence of the technique is to allow programs on different machines to interact using simple procedure call/return semantics, just as if the two programs were on the same machine. That is, the procedure call is used for access to remote services. The popularity of this approach is due to the following advantages.

1. The procedure call is a widely accepted, used, and understood abstraction.
2. The use of remote procedure calls enables remote interfaces to be specified as a set of named operations with designated types. Thus, the interface can be clearly documented, and distributed programs can be statically checked for type errors.
3. Because a standardized and precisely defined interface is specified, the communication code for an application can be generated automatically.
4. Because a standardized and precisely defined interface is specified, developers can write client and server modules that can be moved among computers and operating systems with little modification and recoding.

The remote procedure call mechanism can be viewed as a refinement of reliable, blocking message passing. Figure 18.10b illustrates the general architecture, and Figure 18.12 provides a more detailed look. The calling program makes a normal procedure call with parameters on its machine. For example,

CALL P(X, Y)

where

P = procedure name
X = passed arguments
Y = returned values

It may or may not be transparent to the user that the intention is to invoke a remote procedure on some other machine. A dummy or stub procedure P must be included in the caller's address space or be dynamically linked to it at call time. This procedure creates a message that identifies the procedure being called and includes the parameters. It then sends this message to a remote system and waits for a reply. When a reply is received, the stub procedure returns to the calling program, providing the returned values.

At the remote machine, another stub program is associated with the called procedure. When a message comes in, it is examined and a local CALL P(X, Y) is generated. This remote procedure is thus called locally, so its normal assumptions about where to find parameters, the state of the stack, and so on are identical to the case of a purely local procedure call.

Several design issues are associated with remote procedure calls, and these are addressed in the remainder of this section.

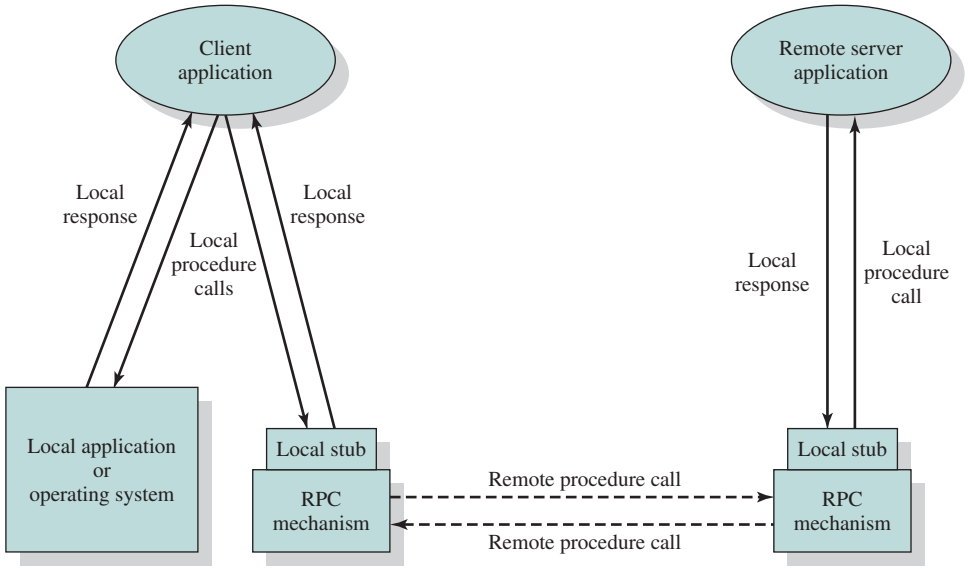


Figure 18.12 **Remote Procedure Call Mechanism**

Parameter Passing

Most programming languages allow parameters to be passed as values (call by value) or as pointers to a location that contains the value (call by reference). Call by value is simple for a remote procedure call: The parameters are simply copied into the message and sent to the remote system. It is more difficult to implement call by reference. A unique, system-wide pointer is needed for each object. The overhead for this capability may not be worth the effort.

Parameter Representation

Another issue is how to represent parameters and results in messages. If the called and calling programs are in identical programming languages on the same type of machines with the same operating system, then the representation requirement may present no problems. If there are differences in these areas, then there will probably be differences in the ways in which numbers and even text are represented. If a full-blown communications architecture is used, then this issue is handled by the presentation layer. However, the overhead of such an architecture has led to the design of remote procedure call facilities that bypass most of the communications architecture and provide their own basic communications facility. In that case, the conversion responsibility falls on the remote procedure call facility (e.g., see [GIBB87]).

The best approach to this problem is to provide a standardized format for common objects, such as integers, floating-point numbers, characters, and character strings. Then the native parameters on any machine can be converted to and from the standardized representation.

Client/Server Binding

Binding specifies how the relationship between a remote procedure and the calling program will be established. A binding is formed when two applications have made a logical connection and are prepared to exchange commands and data.

Nonpersistent binding means that a logical connection is established between the two processes at the time of the remote procedure call, and that as soon as the values are returned, the connection is dismantled. Because a connection requires the maintenance of state information on both ends, it consumes resources. The nonpersistent style is used to conserve those resources. On the other hand, the overhead involved in establishing connections makes nonpersistent binding inappropriate for remote procedures that are called frequently by the same caller.

With **persistent binding**, a connection that is set up for a remote procedure call is sustained after the procedure return. The connection can then be used for future remote procedure calls. If a specified period of time passes with no activity on the connection, then the connection is terminated. For applications that make many repeated calls to remote procedures, persistent binding maintains the logical connection and allows a sequence of calls and returns to use the same connection.

Synchronous versus Asynchronous

The concepts of synchronous and asynchronous remote procedure calls are analogous to the concepts of blocking and nonblocking messages. The traditional remote procedure call is synchronous, which requires that the calling process wait until the called process returns a value. Thus, the **synchronous RPC** behaves much like a subroutine call.

The synchronous RPC is easy to understand and program because its behavior is predictable. However, it fails to exploit fully the parallelism inherent in distributed applications. This limits the kind of interaction the distributed application can have, resulting in lower performance.

To provide greater flexibility, various **asynchronous RPC** facilities have been implemented to achieve a greater degree of parallelism while retaining the familiarity and simplicity of the RPC [ANAN92]. Asynchronous RPCs do not block the caller; the replies can be received as and when they are needed, thus allowing client execution to proceed locally in parallel with the server invocation.

A typical asynchronous RPC use is to enable a client to invoke a server repeatedly so the client has a number of requests in the pipeline at one time, each with its own set of data. Synchronization of client and server can be achieved in one of two ways:

1. A higher-layer application in the client and server can initiate the exchange then check at the end that all requested actions have been performed.
2. A client can issue a string of asynchronous RPCs followed by a final synchronous RPC. The server will respond to the synchronous RPC only after completing all of the work requested in the preceding asynchronous RPCs.

In some schemes, asynchronous RPCs require no reply from the server and the server cannot send a reply message. Other schemes either require or allow a reply, but the caller does not wait for the reply.

Object-Oriented Mechanisms

As object-oriented technology becomes more prevalent in operating system design, client/server designers have begun to embrace this approach. In this approach, clients and servers ship messages back and forth between objects. Object communications may rely on an underlying message or RPC structure or be developed directly on top of object-oriented capabilities in the operating system.

A client that needs a service sends a request to an object request broker, which acts as a directory of all the remote service available on the network (see Figure 18.10c). The broker calls the appropriate object and passes along any relevant data. Then the remote object services the request and replies to the broker, which returns the response to the client.

The success of the object-oriented approach depends on standardization of the object mechanism. Unfortunately, there are several competing designs in this area. One is Microsoft's Component Object Model (COM), the basis for Object Linking and Embedding (OLE). A competing approach, developed by the Object Management Group, is the Common Object Request Broker Architecture (CORBA), which has wide industry support. IBM, Apple, Sun, and many other vendors support the CORBA approach.

18.4 CLUSTERS

Clustering is an alternative to symmetric multiprocessing (SMP) as an approach to providing high performance and high availability and is particularly attractive for server applications. We can define a cluster as a group of interconnected, whole computers working together as a unified computing resource that can create the illusion of being one machine. The term *whole computer* means a system that can run on its own, apart from the cluster; in the literature, each computer in a cluster is typically referred to as a *node*.

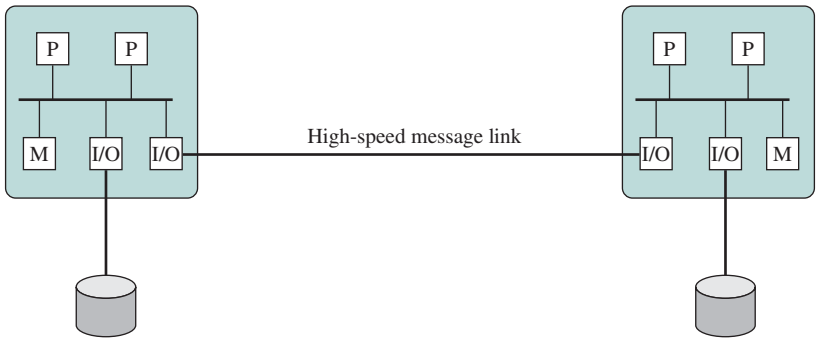
[BREW97] lists four benefits that can be achieved with clustering. These can also be thought of as objectives or design requirements:

- **Absolute scalability:** It is possible to create large clusters that far surpass the power of even the largest stand-alone machines. A cluster can have dozens or even hundreds of machines, each of which is a multiprocessor.
- **Incremental scalability:** A cluster is configured in such a way that it is possible to add new systems to the cluster in small increments. Thus, a user can start out with a modest system and expand it as needs grow, without having to go through a major upgrade in which an existing small system is replaced with a larger system.
- **High availability:** Because each node in a cluster is a stand-alone computer, the failure of one node does not mean loss of service. In many products, fault tolerance is handled automatically in software.
- **Superior price/performance:** By using commodity building blocks, it is possible to put together a cluster with equal or greater computing power than a single large machine, at much lower cost.

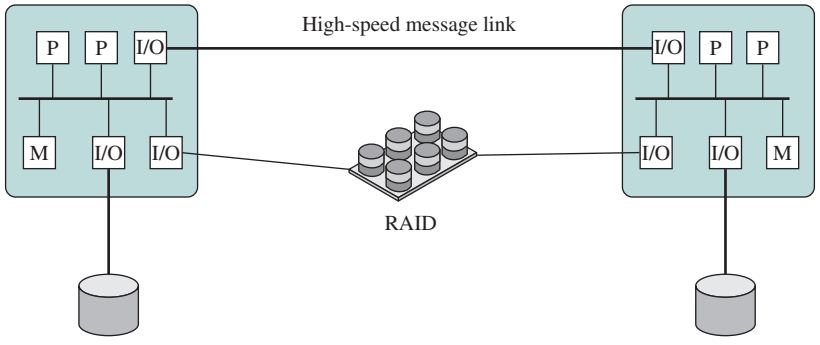
Cluster Configurations

In the literature, clusters are classified in a number of different ways. Perhaps the simplest classification is based on whether the computers in a cluster share access to the same disks. Figure 18.13a shows a two-node cluster in which the only interconnection is by means of a high-speed link that can be used for message exchange to coordinate cluster activity. The link can be a LAN that is shared with other computers that are not part of the cluster, or the link can be a dedicated interconnection facility. In the latter case, one or more of the computers in the cluster will have a link to a LAN or WAN so there is a connection between the server cluster and remote client systems. Note in the figure, each computer is depicted as being a multiprocessor. This is not necessary but does enhance both performance and availability.

In the simple classification depicted in Figure 18.13, the other alternative is a shared disk cluster. In this case, there generally is still a message link between nodes. In addition, there is a disk subsystem that is directly linked to multiple computers within the cluster. In Figure 18.13b, the common disk subsystem is a RAID system. The use of RAID or some similar redundant disk technology is common in clusters so the high availability achieved by the presence of multiple computers is not compromised by a shared disk that is a single point of failure.



(a) Standby server with no shared disk



(b) Shared disk

Figure 18.13 Cluster Configurations

Table 18.2 Clustering Methods: Benefits and Limitations

Clustering Method	Description	Benefits	Limitations
Passive Standby	A secondary server takes over in case of primary server failure.	Easy to implement.	High cost because the secondary server is unavailable for other processing tasks.
Active Secondary	The secondary server is also used for processing tasks.	Reduced cost because secondary servers can be used for processing.	Increased complexity.
Separate Servers	Separate servers have their own disks. Data are continuously copied from primary to secondary server.	High availability.	High network and server overhead due to copying operations.
Servers Connected to Disks	Servers are cabled to the same disks, but each server owns its disks. If one server fails, its disks are taken over by the other server.	Reduced network and server overhead due to elimination of copying operations.	Usually requires disk mirroring or RAID technology to compensate for risk of disk failure.
Servers Share Disks	Multiple servers simultaneously share access to disks.	Low network and server overhead. Reduced risk of downtime caused by disk failure.	Requires lock manager software. Usually used with disk mirroring or RAID technology.

A clearer picture of the range of clustering approaches can be gained by looking at functional alternatives. A white paper from Hewlett Packard [HP96] provides a useful classification along functional lines (see Table 18.2), which we now discuss.

A common, older method, known as **passive standby**, is simply to have one computer handle all of the processing load while the other computer remains inactive, standing by to take over in the event of a failure of the primary. To coordinate the machines, the active, or primary, system periodically sends a "heartbeat" message to the standby machine. Should these messages stop arriving, the standby assumes that the primary server has failed and puts itself into operation. This approach increases availability but does not improve performance. Further, if the only information that is exchanged between the two systems is a heartbeat message, and if the two systems do not share common disks, then the standby provides a functional backup but has no access to the databases managed by the primary.

The passive standby is generally not referred to as a cluster. The term cluster is reserved for multiple interconnected computers that are all actively doing processing while maintaining the image of a single system to the outside world. The term **active secondary** is often used in referring to this configuration. Three classifications of clustering can be identified: separate servers, shared nothing, and shared memory.

In one approach to clustering, each computer is a **separate server** with its own disks and there are no disks shared between systems (see Figure 18.13a). This arrangement provides high performance as well as high availability. In this case, some type of management or scheduling software is needed to assign incoming client requests to servers so the load is balanced and high utilization is achieved. It is desirable to

have a failover capability, which means that if a computer fails while executing an application, another computer in the cluster can pick up and complete the application. For this to happen, data must constantly be copied among systems so each system has access to the current data of the other systems. The overhead of this data exchange ensures high availability at the cost of a performance penalty.

To reduce the communications overhead, most clusters now consist of servers connected to common disks (see Figure 18.13b). In one variation of this approach, called **shared nothing**, the common disks are partitioned into volumes, and each volume is owned by a single computer. If that computer fails, the cluster must be reconfigured so some other computer has ownership of the volumes of the failed computer.

It is also possible to have multiple computers share the same disks at the same time (called the **shared disk** approach), so each computer has access to all of the volumes on all of the disks. This approach requires the use of some type of locking facility to ensure data can only be accessed by one computer at a time.

Operating System Design Issues

Full exploitation of a cluster hardware configuration requires some enhancements to a single-system operating system.

Failure Management How failures are managed by a cluster depends on the clustering method used (see Table 18.2). In general, two approaches can be taken to dealing with failures: highly available clusters and fault-tolerant clusters. A highly available cluster offers a high probability that all resources will be in service. If a failure occurs, such as a node goes down or a disk volume is lost, then the queries in progress are lost. Any lost query, if retried, will be serviced by a different computer in the cluster. However, the cluster operating system makes no guarantee about the state of partially executed transactions. This would need to be handled at the application level.

A fault-tolerant cluster ensures all resources are always available. This is achieved by the use of redundant shared disks and mechanisms for backing out uncommitted transactions and committing completed transactions.

The function of switching an application and data resources over from a failed system to an alternative system in the cluster is referred to as **failover**. A related function is the restoration of applications and data resources to the original system once it has been fixed; this is referred to as **failback**. Failback can be automated, but this is desirable only if the problem is truly fixed and unlikely to recur. If not, automatic failback can cause subsequently failed resources to bounce back and forth between computers, resulting in performance and recovery problems.

Load Balancing A cluster requires an effective capability for balancing the load among available computers. This includes the requirement that the cluster be incrementally scalable. When a new computer is added to the cluster, the load-balancing facility should automatically include this computer in scheduling applications. Middleware mechanisms need to recognize that services can appear on different members of the cluster and may migrate from one member to another.

Parallelizing Computation In some cases, effective use of a cluster requires executing software from a single application in parallel. [KAPP00] lists three general approaches to the problem:

- **Parallelizing compiler:** A parallelizing compiler determines, at compile time, which parts of an application can be executed in parallel. These are then split off to be assigned to different computers in the cluster. Performance depends on the nature of the problem and how well the compiler is designed.
- **Parallelized application:** In this approach, the programmer writes the application from the outset to run on a cluster and uses message passing to move data, as required, between cluster nodes. This places a high burden on the programmer but may be the best approach for exploiting clusters for some applications.
- **Parametric computing:** This approach can be used if the essence of the application is an algorithm or program that must be executed a large number of times, each time with a different set of starting conditions or parameters. A good example is a simulation model, which will run a large number of different scenarios, then develop statistical summaries of the results. For this approach to be effective, parametric processing tools are needed to organize, run, and manage the jobs in an orderly manner.

Cluster Computer Architecture

Figure 18.14 shows a typical cluster architecture. The individual computers are connected by some high-speed LAN or switch hardware. Each computer is capable of operating independently. In addition, a middleware layer of software is installed in each computer to enable cluster operation. The cluster middleware provides a unified system image to the user, known as a **single-system image**. The middleware may also be responsible for providing high availability, by means of load balancing and responding to failures in individual components. [HWAN99] lists the following as desirable cluster middleware services and functions:

- **Single entry point:** A user logs on to the cluster rather than to an individual computer
- **Single file hierarchy:** The user sees a single hierarchy of file directories under the same root directory.
- **Single control point:** There is a default node used for cluster management and control.
- **Single virtual networking:** Any node can access any other point in the cluster, even though the actual cluster configuration may consist of multiple interconnected networks. There is a single virtual network operation.
- **Single memory space:** Distributed shared memory enables programs to share variables.
- **Single job-management system:** Under a cluster job scheduler, a user can submit a job without specifying the host computer to execute the job.

- **Single-user interface:** A common graphic interface supports all users, regardless of the workstation from which they enter the cluster.
- **Single I/O space:** Any node can remotely access any I/O peripheral or disk device without knowledge of its physical location.
- **Single process space:** A uniform process-identification scheme is used. A process on any node can create or communicate with any other process on a remote node.
- **Checkpointing:** This function periodically saves the process state and intermediate computing results, to allow rollback recovery after a failure.
- **Process migration:** This function enables load balancing.

The last four items on the preceding list enhance the availability of the cluster. The remaining items are concerned with providing a single-system image.

Returning to Figure 18.14, a cluster will also include software tools for enabling the efficient execution of programs that are capable of parallel execution.

Clusters Compared to SMP

Both clusters and symmetric multiprocessors provide a configuration with multiple processors to support high-demand applications. Both solutions are commercially available, although SMP has been around far longer.

The main strength of the SMP approach is that an SMP is easier to manage and configure than a cluster. The SMP is much closer to the original single-processor model for which nearly all applications are written. The principal change required in going from a uniprocessor to an SMP is to the scheduler function. Another benefit of the SMP is that it usually takes up less physical space and draws less power than

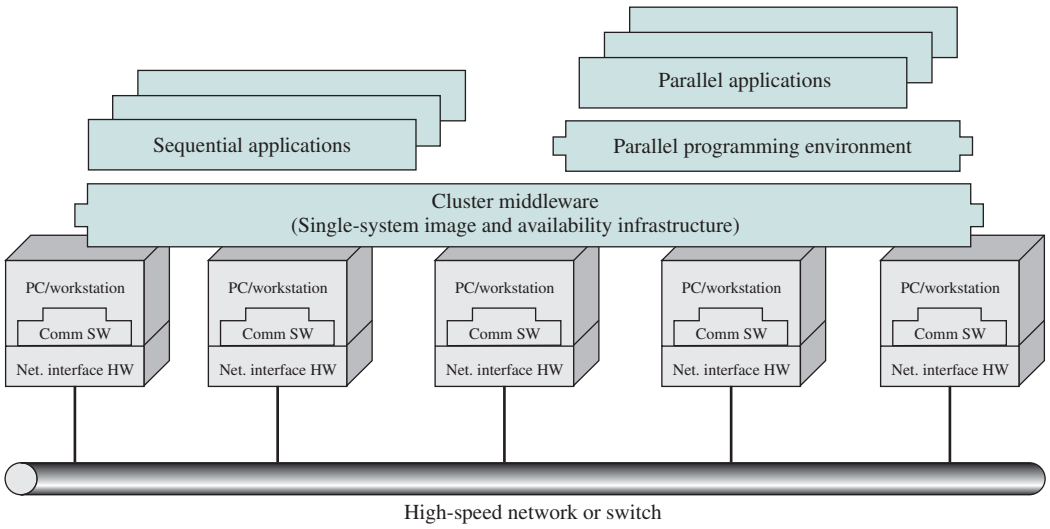


Figure 18.14 Cluster Computer Architecture

a comparable cluster. A final important benefit is that the SMP products are well established and stable.

Over the long run, however, the advantages of the cluster approach are likely to result in clusters dominating the high-performance server market. Clusters are far superior to SMPs in terms of incremental and absolute scalability. Clusters are also superior in terms of availability, because all components of the system can readily be made highly redundant.

18.5 WINDOWS CLUSTER SERVER

Windows Failover Clustering is a shared-nothing cluster, in which each disk volume and other resources are owned by a single system at a time.

The Windows cluster design makes use of the following concepts:

- **Cluster Service:** The collection of software on each node that manages all cluster-specific activity.
- **Resource:** An item managed by the cluster service. All resources are objects representing actual resources in the system, including hardware devices such as disk drives and network cards and logical items such as logical disk volumes, TCP/IP addresses, entire applications, and databases.
- **Online:** A resource is said to be online at a node when it is providing service on that specific node.
- **Group:** A collection of resources managed as a single unit. Usually, a group contains all of the elements needed to run a specific application, and for client systems to connect to the service provided by that application.

The concept of *group* is of particular importance. A group combines resources into larger units that are easily managed, both for failover and load balancing. Operations performed on a group, such as transferring the group to another node, automatically affect all of the resources in that group. Resources are implemented as dynamically linked libraries (DLLs) and managed by a resource monitor. The resource monitor interacts with the cluster service via remote procedure calls and responds to cluster service commands to configure and move resource groups.

Figure 18.15 depicts the Windows clustering components and their relationships in a single system of a cluster. The **node manager** is responsible for maintaining this node's membership in the cluster. Periodically, it sends heartbeat messages to the node managers on other nodes in the cluster. In the event that one node manager detects a loss of heartbeat messages from another cluster node, it broadcasts a message to the entire cluster, causing all members to exchange messages to verify their view of current cluster membership. If a node manager does not respond, it is removed from the cluster and its active groups are transferred to one or more other active nodes in the cluster.

The **configuration database manager** maintains the cluster configuration database. The database contains information about resources and groups and node

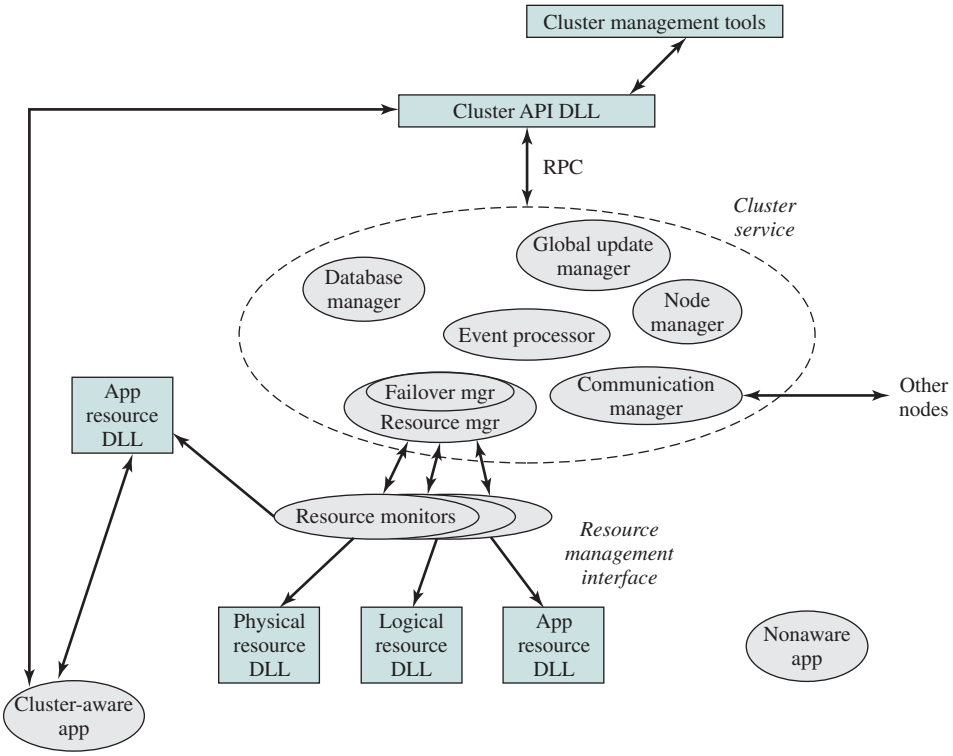


Figure 18.15 Windows Cluster Server Block Diagram

ownership of groups. The database managers on each of the cluster nodes cooperate to maintain a consistent picture of configuration information. Fault-tolerant transaction software is used to assure that changes in the overall cluster configuration are performed consistently and correctly.

The **resource manager/failover manager** makes all decisions regarding resource groups and initiates appropriate actions such as startup, reset, and failover. When failover is required, the failover managers on the active node cooperate to negotiate a distribution of resource groups from the failed system to the remaining active systems. When a system restarts after a failure, the failover manager can decide to move some groups back to this system. In particular, any group may be configured with a preferred owner. If that owner fails and then restarts, the group is moved back to the node in a rollback operation.

The **event processor** connects all of the components of the cluster service, handles common operations, and controls cluster service initialization. The communications manager manages message exchange with all other nodes of the cluster. The global update manager provides a service used by other components within the cluster service.

Microsoft is continuing to ship their cluster product, but they have also developed virtualization solutions based on efficient live migration of virtual

machines between hypervisors running on different computer systems as part of Windows Server 2008 R2. For new applications, live migration offers many benefits over the cluster approach, such as simpler management, and improved flexibility.

18.6 BEOWULF AND LINUX CLUSTERS

In 1994, the Beowulf project was initiated under the sponsorship of the NASA High Performance Computing and Communications (HPCC) project. Its goal was to investigate the potential of clustered PCs for performing important computation tasks beyond the capabilities of contemporary workstations at minimum cost. Today, the Beowulf approach is widely implemented and is perhaps the most important cluster technology available.

Beowulf Features

Key features of Beowulf include the following [RIDG97]:

- Mass market commodity components
- Dedicated processors (rather than scavenging cycles from idle workstations)
- A dedicated, private network (LAN or WAN or internetted combination)
- No custom components
- Easy replication from multiple vendors
- Scalable I/O
- A freely available software base
- Use of freely available distribution computing tools with minimal changes
- Return of the design and improvements to the community

Although elements of Beowulf software have been implemented on a number of different platforms, the most obvious choice for a base is Linux, and most Beowulf implementations use a cluster of Linux workstations and/or PCs. Figure 18.16 depicts a representative configuration. The cluster consists of a number of workstations, perhaps of differing hardware platforms, all running the Linux operating system. Secondary storage at each workstation may be made available for distributed access (for distributed file sharing, distributed virtual memory, or other uses). The cluster nodes (the Linux systems) are interconnected with a commodity networking approach, typically Ethernet. The Ethernet support may be in the form of a single Ethernet switch or an interconnected set of switches. Commodity Ethernet products at the standard data rates (10 Mbps, 100 Mbps, 1 Gbps) are used.

Beowulf Software

The Beowulf software environment is implemented as an add-on to commercially available, royalty-free base Linux distributions. The principal source of open-source Beowulf software is the Beowulf site at www.beowulf.org, but numerous other organizations also offer free Beowulf tools and utilities.

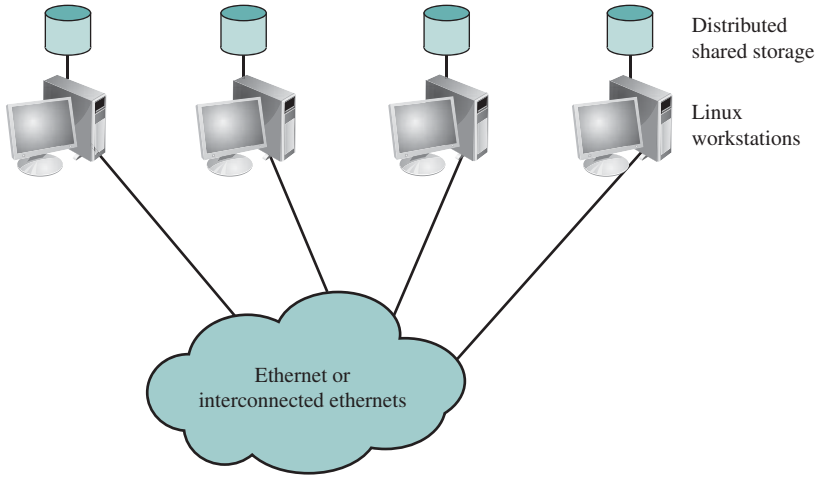


Figure 18.16 Generic Beowulf Configuration

Each node in the Beowulf cluster runs its own copy of the Linux kernel and can function as an autonomous Linux system. To support the Beowulf cluster concept, extensions are made to the Linux kernel to allow the individual nodes to participate in a number of global namespaces. The following are examples of Beowulf system software:

- **Beowulf distributed process space (BPROC):** This package allows a process ID space to span multiple nodes in a cluster environment and also provides mechanisms for starting processes on other nodes. The goal of this package is to provide key elements needed for a single-system image on Beowulf cluster. BPROC provides a mechanism to start processes on remote nodes without ever logging into another node, and by making all the remote processes visible in the process table of the cluster's front-end node.
- **Beowulf Ethernet channel bonding:** This is a mechanism that joins multiple low-cost networks into a single logical network with higher bandwidth. The only additional work over using single network interface is the computationally simple task of distributing the packets over the available device transmit queues. This approach allows load balancing over multiple Ethernets connected to Linux workstations.
- **Pvmsync:** This is a programming environment that provides synchronization mechanisms and shared data objects for processes in a Beowulf cluster.
- **EnFuzion:** EnFuzion consists of a set of tools for doing parametric computing. Parametric computing involves the execution of a program as a large number of jobs, each with different parameters or starting conditions. EnFusion emulates a set of robot users on a single root node machine, each of which will log into one of the many clients that form a cluster. Each job is set up to run with a unique, programmed scenario, with an appropriate set of starting conditions [KAPP00].

18.7 SUMMARY

Client/server computing is the key to realizing the potential of information systems and networks to improve productivity significantly in organizations. With client/server computing, applications are distributed to users on single-user workstations and personal computers. At the same time, resources that can and should be shared are maintained on server systems that are available to all clients. Thus, the client/server architecture is a blend of decentralized and centralized computing.

Typically, the client system provides a graphical user interface (GUI) that enables a user to exploit a variety of applications with minimal training and relative ease. Servers support shared utilities, such as database management systems. The actual application is divided between client and server in a way intended to optimize ease of use and performance.

The key mechanism required in any distributed system is interprocess communication. Two techniques are in common use. A message-passing facility generalizes the use of messages within a single system. The same sorts of conventions and synchronization rules apply. Another approach is the use of the remote procedure call. This is a technique by which two programs on different machines interact using procedure call/return syntax and semantics. Both the called and calling program behave as if the partner program were running on the same machine.

A cluster is a group of interconnected, whole computers working together as a unified computing resource that can create the illusion of being one machine. The term *whole computer* means a system that can run on its own, apart from the cluster.

18.8 REFERENCES

ANAN92 Ananda, A.; Tay, B.; and Koh, E. "A Survey of Asynchronous Remote Procedure Calls." *Operating Systems Review*, April 1992.

BREW97 Brewer, E. "Clustering: Multiply and Conquer." *Data Communications*, July 1997.

GIBB87 Gibbons, P. "A Stub Generator for Multilanguage RPC in Heterogeneous Environments." *IEEE Transactions on Software Engineering*, January 1987.

HP96 Hewlett Packard. *White Paper on Clustering*. June 1996.

HWAN99 Hwang, K., et al. "Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space." *IEEE Concurrency*, January–March 1999.

KAPP00 Kapp, C. "Managing Cluster Computers." *Dr. Dobb's Journal*, July 2000.

NELS88 Nelson, M.; Welch, B.; and Ousterhout, J. "Caching in the Sprite Network File System." *ACM Transactions on Computer Systems*, February 1988.

OUST88 Ousterhout, J., et al. "The Sprite Network Operating System." *Computer*, February 1988.

RIDG97 Ridge, D., et al. "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs." *Proceedings, IEEE Aerospace Conference*, 1997.

18.9 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

applications programming interface	failback	middleware
Beowulf	failover	remote procedure call (RPC)
client	fat client	server
cluster	file cache consistency	thin client
distributed message passing	graphical user interface (GUI)	
	message	

Review Questions

18.1. What is client/server computing?

18.2. What distinguishes client/server computing from any other form of distributed data processing?

18.3. What is the role of a communications architecture such as TCP/IP in a client/server environment?

18.4. Discuss the rationale for locating applications on the client, the server, or split between client and server.

18.5. What are fat clients and thin clients, and what are the differences in philosophy of the two approaches?

18.6. Suggest pros and cons for fat client and thin client strategies.

18.7. Explain the rationale behind the three-tier client/server architecture.

18.8. What is middleware?

18.9. Because we have standards such as TCP/IP, why is middleware needed?

18.10. List some benefits and disadvantages of blocking and nonblocking primitives for message passing.

18.11. List some benefits and disadvantages of nonpersistent and persistent binding for RPCs.

18.12. List some benefits and disadvantages of synchronous and asynchronous RPCs.

18.13. List and briefly define four different clustering methods.

Problems

18.1. Let α be the percentage of program code that can be executed simultaneously by n computers in a cluster, each computer using a different set of parameters or initial conditions. Assume the remaining code must be executed sequentially by a single processor. Each processor has an execution rate of x MIPS.

a. Derive an expression for the effective MIPS rate when using the system for exclusive execution of this program, in terms of n, α, and x.

b. If $n = 16$ and $x = 4$ MIPS, determine the value of α that will yield a system performance of 40 MIPS.

18.2. An application program is executed on a nine-computer cluster. A benchmark program takes time T on this cluster. Further, 25% of T is time in which the application is running simultaneously on all nine computers. The remaining time, the application has to run on a single computer.

a. Calculate the effective speedup under the aforementioned condition as compared to executing the program on a single computer. Also calculate, the percentage of code that has been parallelized (programmed or compiled so as to use the cluster mode) in the preceding program.

b. Suppose we are able to effectively use 18 computers rather than 9 computers on the parallelized portion of the code. Calculate the effective speedup that is achieved.

18.3. The following FORTRAN program is to be executed on a computer, and a parallel version is to be executed on a 32-computer cluster:

```
L1:      DO 10 I = 1,1024
L2:      SUM(I) = 0
L3:      DO 20 J = 1, I
L4: 20 SUM(I) = SUM(I) + I
L5: 10 CONTINUE
```

Suppose lines 2 and 4 each take two machine cycle times, including all processor and memory-access activities. Ignore the overhead caused by the software loop control statements (lines 1, 3, 5) and all other system overhead and resource conflicts.

a. What is the total execution time (in machine cycle times) of the program on a single computer?

b. Divide the I-loop iterations among the 32 computers as follows: Computer 1 executes the first 32 iterations (I = 1 to 32), processor 2 executes the next 32 iterations, and so on. What are the execution time and speedup factor compared with part (a)? (Note the computational workload, dictated by the J-loop, is unbalanced among the computers.)

c. Explain how to modify the parallelizing to facilitate a balanced parallel execution of all the computational workload over 32 computers. A balanced load means an equal number of additions assigned to each computer with respect to both loops.

d. What is the minimum execution time resulting from the parallel execution on 32 computers? What is the resulting speedup over a single computer?

CHAPTER 19

Distributed Process Management

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Give an explanation of process migration.
- Understand the concept of distributed global states.
- Analyze distributed mutual exclusion algorithms.
- Analyze distributed deadlock algorithms.

This chapter examines key mechanisms used in distributed operating systems. First we look at process migration, which is the movement of an active process from one machine to another. Next, we examine the question of how processes on different systems can coordinate their activities when each is governed by a local clock and when there is a delay in the exchange of information. Finally, we explore two key issues in distributed process management: mutual exclusion and deadlock.

19.1 PROCESS MIGRATION

Process migration is the transfer of a sufficient amount of the state of a process from one computer to another for the process to execute on the target machine. Interest in this concept grew out of research into methods of load balancing across multiple networked systems, although the application of the concept now extends beyond that one area.

In the past, only a few of the many papers on load distribution were based on true implementations of process migration, which includes the ability to preempt a process on one machine and reactivate it later on another machine. Experience showed that preemptive process migration is possible, although with higher overhead and complexity than originally anticipated [ARTS89a]. This cost led some observers to conclude that process migration was not practical. Such assessments have proved too pessimistic. New implementations, including those in commercial products, have fueled a continuing interest and new developments in this area. This section provides an overview.

Motivation

Process migration is desirable in distributed systems for a number of reasons [SMIT88, JUL88], including:

- **Load sharing:** By moving processes from heavily loaded to lightly loaded systems, the load can be balanced to improve overall performance. Empirical data suggest that significant performance improvements are possible [LELA86, CABR86]. However, care must be taken in the design of load-balancing algorithms. [EAGE86] points out that the more communication necessary for the distributed system to perform the balancing, the worse the performance becomes. A discussion of this issue, with references to other studies, can be found in [ESKI90].

- **Communications performance:** Processes that interact intensively can be moved to the same node to reduce communications cost for the duration of their interaction. Also, when a process is performing data analysis on some file or set of files larger than the process's size, it may be advantageous to move the process to the data rather than vice versa.
- **Availability:** Long-running processes may need to move to survive in the face of faults for which advance notice is possible or in advance of scheduled downtime. If the operating system provides such notification, a process that wants to continue can either migrate to another system or ensure that it can be restarted on the current system at some later time.
- **Utilizing special capabilities:** A process can move to take advantage of unique hardware or software capabilities on a particular node.

Process Migration Mechanisms

A number of issues need to be addressed in designing a process migration facility. Among these are the following:

- Who initiates the migration?
- What portion of the process is migrated?
- What happens to outstanding messages and signals?

Initiation of Migration Who initiates migration will depend on the goal of the migration facility. If the goal is load balancing, then some module in the operating system that is monitoring system load will generally be responsible for deciding when migration should take place. The module will be responsible for preempting or signaling a process to be migrated. To determine where to migrate, the module will need to be in communication with peer modules in other systems so the load patterns on other systems can be monitored. If the goal is to reach particular resources, then a process may migrate itself as the need arises. In this latter case, the process must be aware of the existence of a distributed system. In the former case, the entire migration function, and indeed the existence of multiple systems, may be transparent to the process.

What Is Migrated? When a process is migrated, it is necessary to destroy the process on the source system and create it on the target system. This is a movement of a process, not a replication. Thus, the process image, consisting of at least the process control block, must be moved. In addition, any links between this process and other processes, such as for passing messages and signals, must be updated. Figure 19.1 illustrates these considerations. Process 3 has migrated out of machine S to become Process 4 in machine D. All link identifiers held by processes (denoted in lowercase letters) remain the same as before. It is the responsibility of the operating system to move the process control block and to update link mappings. The transfer of the process of one machine to another is invisible to both the migrated process and its communication partners.

The movement of the process control block is straightforward. The difficulty, from a performance point of view, concerns the process address space and any open

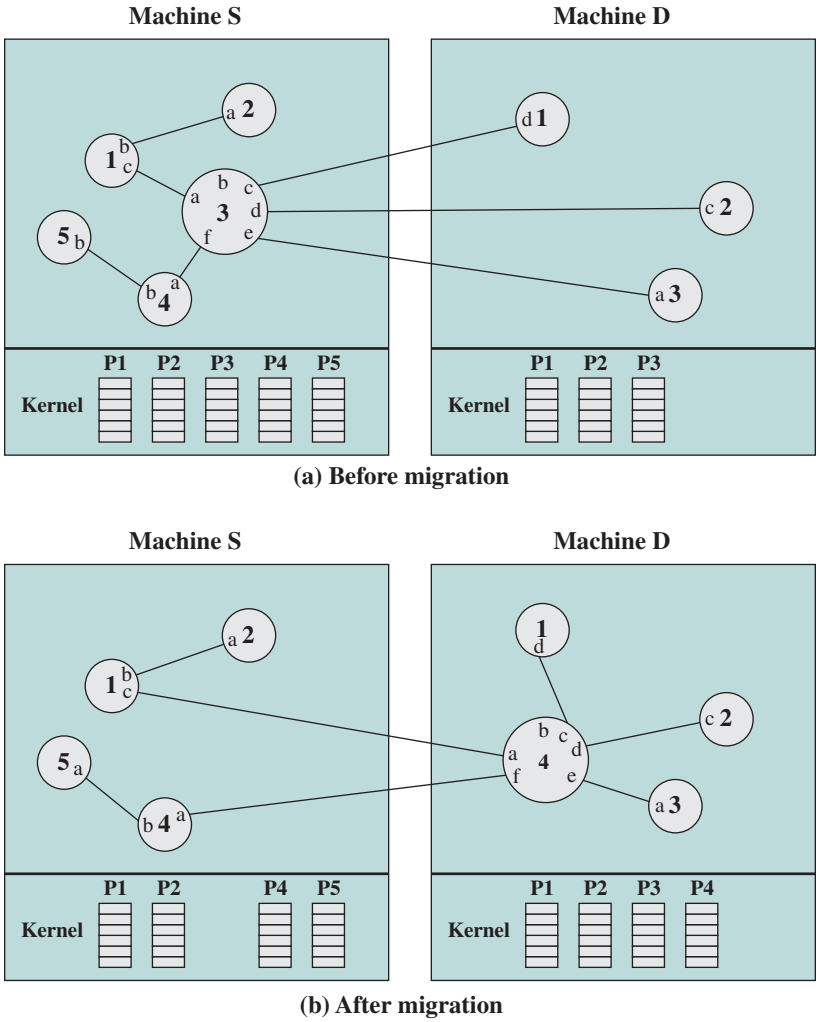


Figure 19.1 **Example of Process Migration**

files assigned to the process. Consider first the process address space and let us assume that a virtual memory scheme (paging or paging/segmentation) is being used. The following strategies have been considered [MILO00]:

- **Eager (all):** Transfer the entire address space at the time of migration. This is certainly the cleanest approach. No trace of the process need to be left behind at the old system. However, if the address space is very large and if the process is likely not to need most of it, then this may be unnecessarily expensive. Initial costs of migration may be on the order of minutes. Implementations that provide a checkpoint/restart facility are likely to use this approach, because it is simpler to do the checkpointing and restarting if all of the address space is localized.

- **Precopy:** The process continues to execute on the source node while the address space is copied to the target node. Pages modified on the source during the precopy operation have to be copied a second time. This strategy reduces the time that a process is frozen and cannot execute during migration.
- **Eager (dirty):** Transfer only those pages of the address space that are in main memory and have been modified. Any additional blocks of the virtual address space will be transferred on demand only. This minimizes the amount of data that are transferred. It does require, however, that the source machine continue to be involved in the life of the process by maintaining page and/or segment table entries and it requires remote paging support.
- **Copy-on-reference:** This is a variation of eager (dirty) in which pages are only brought over when referenced. This has the lowest initial cost of process migration, ranging from a few tens to a few hundreds of microseconds.
- **Flushing:** The pages of the process are cleared from the main memory of the source by flushing dirty pages to disk. Then pages are accessed as needed from disk instead of from memory on the source node. This strategy relieves the source of the need to hold any pages of the migrated process in main memory, immediately freeing a block of memory to be used for other processes.

If it is likely that the process will not use much of its address space while on the target machine (e.g., the process is only temporarily going to another machine to work on a file and will soon return), then one of the last three strategies makes sense. On the other hand, if much of the address space will eventually be accessed while on the target machine, then the piecemeal transfer of blocks of the address space may be less efficient than simply transferring all of the address space at the time of migration, using one of the first two strategies.

In many cases, it may not be possible to know in advance whether or not much of the nonresident address space will be needed. However, if processes are structured as threads, and if the basic unit of migration is the thread rather than the process, then a strategy based on remote paging would seem to be the best. Indeed, such a strategy is almost mandated, because the remaining threads of the process are left behind and also need access to the address space of the process. Thread migration is implemented in the Emerald operating system [JUL89].

Similar considerations apply to the movement of open files. If the file is initially on the same system as the process to be migrated, and if the file is locked for exclusive access by that process, then it may make sense to transfer the file with the process. The danger here is that the process may only be gone temporarily and may not need the file until its return. Therefore, it may make sense to transfer the entire file only after an access request is made by the migrated process. If a file is shared by multiple distributed processes, then distributed access to the file should be maintained without moving the file.

If caching is permitted, as in the Sprite system (see Figure 16.7), then an additional complexity is introduced. For example, if a process has a file open for writing and it forks and migrates a child, the file would then be open for writing on two different hosts; Sprite's cache consistency algorithm dictates that the file be made noncacheable on the machines on which the two processes are executing [DOUG89, DOUG91].

Messages and Signals The final issue listed previously, the fate of messages and signals, is addressed by providing a mechanism for temporarily storing outstanding messages and signals during the migration activity then directing them to the new destination. It may be necessary to maintain forwarding information at the initial site for some time to assure that all outstanding messages and signals get through.

A Migration Scenario As a representative example of self-migration, let us consider the facility available on IBM's AIX operating system [WALK89], which is a distributed UNIX operating system. A similar facility is available on the LOCUS operating system [POPE85], and in fact the AIX system is based on the LOCUS development. This facility has also been ported to the OSF/1 AD operating system, under the name TNC [ZAJC93].

The following sequence of events occurs:

1. When a process decides to migrate itself, it selects a target machine and sends a remote tasking message. The message carries a part of the process image and open file information.
2. At the receiving site, a kernel server process forks a child, giving it this information.
3. The new process pulls over data, environment, arguments, or stack information as needed to complete its operation. Program text is copied over if it is dirty or demand paged from the global file system if it is clean.
4. The originating process is signaled on the completion of the migration. This process sends a final done message to the new process and destroys itself.

A similar sequence would be followed when another process initiates the migration. The principal difference is that the process to be migrated must be suspended so it can be migrated in a nonrunning state. This procedure is followed in Sprite, for example [DOUG89].

In the foregoing scenario, migration is a dynamic activity involving a number of steps for moving the process image over. When migration is initiated by another process, rather than self-migration, another approach is to copy the process image and its entire address space into a file, destroy the process, copy the file to another machine using a file transfer facility, then recreate the process from the file on the target machine. [SMIT89] describes such an approach.

Negotiation of Migration

Another aspect of process migration relates to the decision about migration. In some cases, the decision is made by a single entity. For example, if load balancing is the goal, a load-balancing module monitors the relative load on various machines and performs migration as necessary to maintain a load balance. If self-migration is used to allow a process access to special facilities or to large remote files, then the process itself may make the decision. However, some systems allow the designated target system to participate in the decision. One reason for this could be to preserve response time for users. A user at a workstation, for example, might suffer noticeable response time degradation if processes migrate to the user's system, even if such migration served to provide better overall balance.

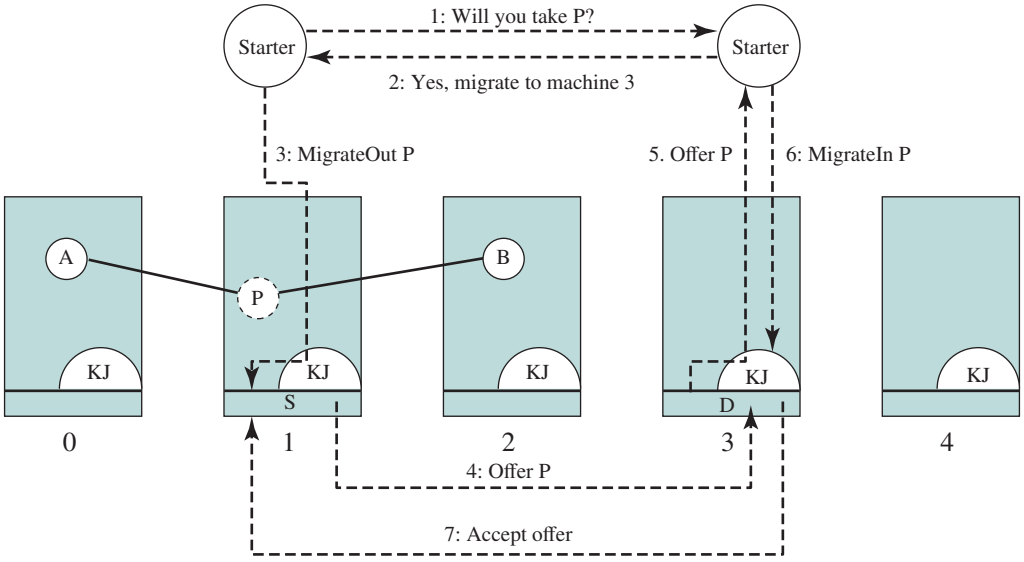


Figure 19.2 **Negotiation of Process Migration**

An example of a negotiation mechanism is that found in Charlotte [FINK89, ARTS89b]. Migration policy (when to migrate which process to what destination) is the responsibility of the Starter utility, which is a process that is also responsible for long-term scheduling and memory allocation. The Starter can therefore coordinate policy in these three areas. Each Starter process may control a cluster of machines. The Starter receives timely and fairly elaborate load statistics from the kernel of each of its machines.

The decision to migrate must be reached jointly by two Starter processes (one on the source node and one on the destination node), as illustrated in Figure 19.2. The following steps occur:

1. The Starter that controls the source system (S) decides that a process P should be migrated to a particular destination system (D). It sends a message to D's Starter, requesting the transfer.
2. If D's Starter is prepared to receive the process, it sends back a positive acknowledgment.
3. S's Starter communicates this decision to S's kernel via service call (if the starter runs on S) or a message to the KernJob (KJ) of machine S (if the starter runs on another machine). KJ is a process used to convert messages from remote processes into service calls.
4. The kernel on S then offers to send the process to D. The offer includes statistics about P, such as its age and processor and communication loads.
5. If D is short of resources, it may reject the offer. Otherwise, the kernel on D relays the offer to its controlling Starter. The relay includes the same information as the offer from S.

6. The Starter's policy decision is communicated to D by a MigrateIn call.
7. D reserves necessary resources to avoid deadlock and flow-control problems and then sends an acceptance to S.

Figure 19.2 also shows two other processes, A and B, that have links open to P. Following the foregoing steps, machine 1, where S resides, must send a link update message to both machines 0 and 2 to preserve the links from A and B to P. Link update messages tell the new address of each link held by P and are acknowledged by the notified kernels for synchronization purposes. After this point, a message sent to P on any of its links will be sent directly to D. These messages can be exchanged concurrently with the steps just described. Finally, after step 7 and after all links have been updated, S collects all of P's context into a single message and sends it to D.

Machine 4 is also running Charlotte but is not involved in this migration and therefore has no communication with the other systems in this episode.

Eviction

The negotiation mechanism allows a destination system to refuse to accept the migration of a process to itself. In addition, it might also be useful to allow a system to evict a process that has been migrated to it. For example, if a workstation is idle, one or more processes may be migrated to it. Once the user of that workstation becomes active, it may be necessary to evict the migrated processes to provide adequate response time.

An example of an eviction capability is that found in Sprite [DOUG89]. In Sprite, which is a workstation operating system, each process appears to run on a single host throughout its lifetime. This host is known as the home node of the process. If a process is migrated, it becomes a foreign process on the destination machine. At any time the destination machine may evict the foreign process, which is then forced to migrate back to its home node.

The elements of the Sprite eviction mechanism are as follows:

1. A monitor process at each node keeps track of current load to determine when to accept new foreign processes. If the monitor detects activity at the workstation's console, it initiates an eviction procedure on each foreign process.
2. If a process is evicted, it is migrated back to its home node. The process may be migrated again if another node is available.
3. Although it may take some time to evict all processes, all processes marked for eviction are immediately suspended. Permitting an evicted process to execute while it is waiting for eviction would reduce the time during which the process is frozen, but also reduce the processing power available to the host while evictions are underway.
4. The entire address space of an evicted process is transferred to the home node. The time to evict a process and migrate it back to its home node may be reduced substantially by retrieving the memory image of an evicted process from its previous foreign host as referenced. However, this compels the foreign host to dedicate resources and honor service requests from the evicted process for a longer period of time than necessary.

Preemptive versus Nonpreemptive Transfers

The discussion in this section has dealt with preemptive process migration, which involves transferring a partially executed process, or at least a process whose creation has been completed. A simpler function is nonpreemptive process transfer, which involves only processes that have not begun execution and hence do not require transferring the state of the process. In both types of transfer, information about the environment in which the process will execute must be transferred to the remote node. This may include the user's current working directory, the privileges inherited by the process, and inherited resources such as file descriptions.

Nonpreemptive process migration can be useful in load balancing (e.g., see [SHIV92]). It has the advantage that it avoids the overhead of full-blown process migration. The disadvantage is that such a scheme does not react well to sudden changes in load distribution.

19.2 DISTRIBUTED GLOBAL STATES

Global States and Distributed Snapshots

All of the concurrency issues that are faced in a tightly coupled system, such as mutual exclusion, deadlock, and starvation, are also faced in a distributed system. Design strategies in these areas are complicated by the fact that there is no global state to the system. That is, it is not possible for the operating system, or any process, to know the current state of all processes in the distributed system. A process can only know the current state of all the processes on the local system, by access to process control blocks in memory. For remote processes, a process can only know state information that is received via messages, which represent the state of the remote process sometime in the past. This is analogous to the situation in astronomy: Our knowledge of a distant star or galaxy consists of light and other electromagnetic waves arriving from the distant object, and these waves provide a picture of the object sometime in the past. For example, our knowledge of an object at a distance of five light-years is five years old.

The time lags imposed by the nature of distributed systems complicate all issues relating to concurrency. To illustrate this, we present an example taken from [ANDR90]. We will use process/event graphs (see Figures 19.3 and 19.4) to illustrate the problem. In these graphs, there is a horizontal line for each process representing the time axis. A point on the line corresponds to an event (e.g., internal process event, message send, message receive). A box surrounding a point represents a snapshot of the local process state taken at that point. An arrow represents a message between two processes.

In our example, an individual has a bank account distributed over two branches of a bank. To determine the total amount in the customer's account, the bank must determine the amount in each branch. Suppose the determination is to be made at exactly 3:00 p.m. Figure 19.3a shows an instance in which a balance of $100.00 in the combined account is found. But the situation in Figure 19.3b is also possible. Here, the balance from branch A is in transit to branch B at the time of observation; the result is a false reading of $0.00. This particular problem can be solved by examining all messages in transit at the time of observation. Branch A will keep a record of all

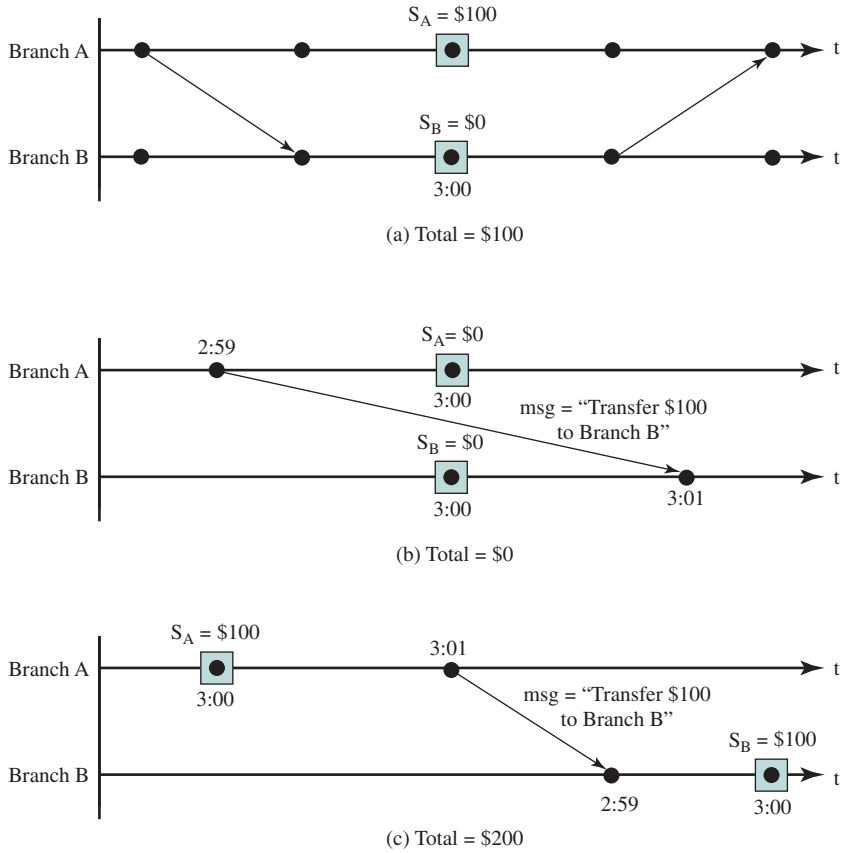


Figure 19.3 Example of Determining Global States

transfers out of the account, together with the identity of the destination of the transfer. Therefore, we will include in the "state" of a branch A account both the current balance and a record of transfers. When the two accounts are examined, the observer finds a transfer that has left branch A destined for the customer's account in branch B. Because the amount has not yet arrived at branch B, it is added into the total balance. Any amount that has been both transferred and received is counted only once, as part of the balance at the receiving account.

This strategy is not foolproof, as shown in Figure 19.3c. In this example, the clocks at the two branches are not perfectly synchronized. The state of the customer account at branch A at 3:00 p.m. indicates a balance of $100.00. However, this amount is subsequently transferred to branch B at 3:01 according to the clock at A but arrives at B at 2:59 according to B's clock. Therefore, the amount is counted twice for a 3:00 observation.

To understand the difficulty we face and to formulate a solution, let us define the following terms:

- **Channel:** A channel exists between two processes if they exchange messages. We can think of the channel as the path or means by which the messages are

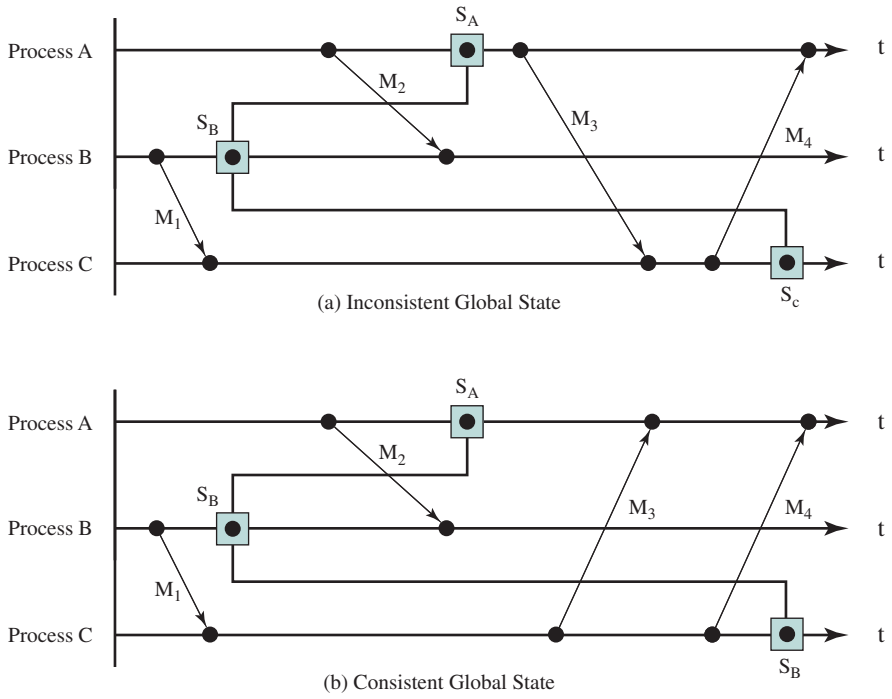


Figure 19.4 Inconsistent and Consistent Global States

transferred. For convenience, channels are viewed as unidirectional. Thus, if two processes exchange messages, two channels are required, one for each direction of message transfer.

- **State:** The state of a process is the sequence of messages that have been sent and received along channels incident with the process.
- **Snapshot:** A snapshot records the state of a process. Each snapshot includes a record of all messages sent and received on all channels since the last snapshot.
- **Global state:** The combined state of all processes.
- **Distributed snapshot:** A collection of snapshots, one for each process.

The problem is that a true global state cannot be determined because of the time lapse associated with message transfer. We can attempt to define a global state by collecting snapshots from all processes. For example, the global state of Figure 19.4a at the time of the taking of snapshots shows a message in transit on the <A,B> channel, one in transit on the <A,C> channel, and one in transit on the <C,A> channel. Messages 2 and 4 are represented appropriately, but message 3 is not. The distributed snapshot indicates that this message has been received but not yet sent.

We desire that the distributed snapshot record a consistent global state. A global state is consistent if for every process state that records the receipt of a

message, the sending of that message is recorded in the process state of the process that sent the message. Figure 19.4b gives an example. An inconsistent global state arises if a process has recorded the receipt of a message but the corresponding sending process has not recorded that the message has been sent (see Figure 19.4a).

The Distributed Snapshot Algorithm

A distributed snapshot algorithm that records a consistent global state has been described in [CHAN85]. The algorithm assumes that messages are delivered in the order in which they are sent, and that no messages are lost. A reliable transport protocol (e.g., TCP) satisfies these requirements. The algorithm makes use of a special control message, called a **marker**.

Some process initiates the algorithm by recording its state and sending a marker on all outgoing channels before any more messages are sent. Each process p then proceeds as follows. Upon the first receipt of the marker (say from process q), receiving process p performs the following:

1. p records its local state S_p.
2. p records the state of the incoming channel from q to p as empty.
3. p propagates the marker to all of its neighbors along all outgoing channels.

These steps must be performed atomically; that is, no messages can be sent or received by p until all 3 steps are performed.

At any time after recording its state, when p receives a marker from another incoming channel (say from process r), it performs the following:

- p records the state of the channel from r to p as the sequence of messages p has received from r from the time p recorded its local state S_p to the time it received the marker from r.

The algorithm terminates at a process once the marker has been received along every incoming channel.

[ANDR90] makes the following observations about the algorithm:

1. Any process may start the algorithm by sending out a marker. In fact, several nodes could independently decide to record the state and the algorithm would still succeed.
2. The algorithm will terminate in finite time if every message (including marker messages) is delivered in finite time.
3. This is a distributed algorithm: Each process is responsible for recording its own state and the state of all incoming channels.
4. Once all of the states have been recorded (the algorithm has terminated at all processes), the consistent global state obtained by the algorithm can be assembled at every process by having every process send the state data that it has recorded along every outgoing channel, and having every process forward the state data that it receives along every outgoing channel. Alternatively, the initiating process could poll all processes to acquire the global state.
5. The algorithm does not affect and is not affected by any other distributed algorithm that the processes are participating in.

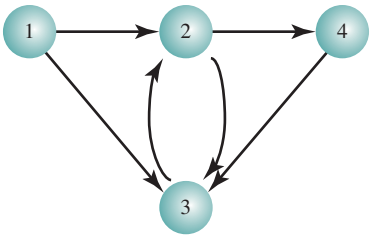


Figure 19.5 **Process and Channel Graph**

As an example of the use of the algorithm (taken from [BEN06]), consider the set of processes illustrated in Figure 19.5. Each process is represented by a node, and each unidirectional channel is represented by a line between two nodes, with the direction indicated by an arrowhead. Suppose the snapshot algorithm is run, with nine messages being sent along each of its outgoing channels by each process. Process 1 decides to record the global state after sending six messages, and process 4 independently decides to record the global state after sending three messages. Upon termination, the snapshots are collected from each process; the results are shown in Figure 19.6. Process 2 sent four messages on each of the two outgoing channels to processes 3 and 4 prior to the recording of the state. It received four messages from process 1 before recording its state, leaving messages 5 and 6 to be associated with the channel. The reader should check the snapshot for consistency: Each message sent was either received at the destination process or recorded as being in transit in the channel.

The distributed snapshot algorithm is a powerful and flexible tool. It can be used to adapt any centralized algorithm to a distributed environment, because the basis of any centralized algorithm is knowledge of the global state. Specific examples include detection of deadlock and detection of process termination (e.g., see [BEN06], [LYNC96]). It can also be used to provide a checkpoint of a distributed algorithm to allow rollback and recovery if a failure is detected.

Process 1 Outgoing channels 2 sent 1,2,3,4,5,6 3 sent 1,2,3,4,5,6 Incoming channels	**Process 3** Outgoing channels 2 sent 1,2,3,4,5,6,7,8 Incoming channels 1 received 1,2,3 stored 4,5,6 2 received 1,2,3 stored 4 4 received 1,2,3
Process 2 Outgoing channels 3 sent 1,2,3,4 4 sent 1,2,3,4 Incoming channels 1 received 1,2,3,4 stored 5,6 3 received 1,2,3,4,5,6,7,8	**Process 4** Outgoing channels 3 sent 1,2,3 Incoming channels 2 received 1,2 stored 3,4

Figure 19.6 **An Example of a Snapshot**

19.3 DISTRIBUTED MUTUAL EXCLUSION

Recall that in Chapters 5 and 6, we addressed issues relating to the execution of concurrent processes. Two key problems that arose were those of mutual exclusion and deadlock. Chapters 5 and 6 focused on solutions to this problem in the context of a single system, with one or more processors but with a common main memory. In dealing with a distributed operating system and a collection of processors that do not share common main memory or clock, new difficulties arise and new solutions are called for. Algorithms for mutual exclusion and deadlock must depend on the exchange of messages and cannot depend on access to common memory. In this section and the next, we examine mutual exclusion and deadlock in the context of a distributed operating system.

Distributed Mutual Exclusion Concepts

When two or more processes compete for the use of system resources, there is a need for a mechanism to enforce mutual exclusion. Suppose two or more processes require access to a single nonsharable resource, such as a printer. During the course of execution, each process will be sending commands to the I/O device, receiving status information, sending data, and/or receiving data. We will refer to such a resource as a critical resource, and the portion of the program that uses it as a critical section of the program. It is important that only one program at a time be allowed in its critical section. We cannot simply rely on the operating system to understand and enforce this restriction, because the detailed requirement may not be obvious. In the case of the printer, for example, we wish any individual process to have control of the printer while it prints an entire file. Otherwise, lines from competing processes will be interleaved.

The successful use of concurrency among processes requires the ability to define critical sections and enforce mutual exclusion. This is fundamental for any concurrent processing scheme. Any facility or capability that is to provide support for mutual exclusion should meet the following requirements:

1. Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its noncritical section must do so without interfering with other processes.
3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or number of processors.
6. A process remains inside its critical section for a finite time only.

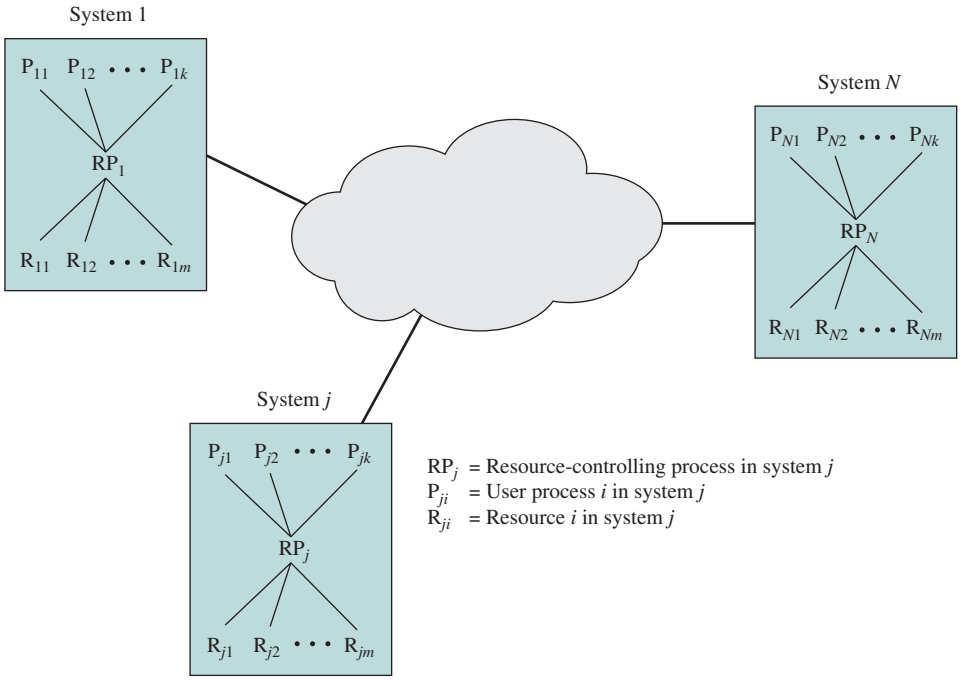


Figure 19.7 Model for Mutual Exclusion Problem in Distributed Process Management

Figure 19.7 shows a model that we can use for examining approaches to mutual exclusion in a distributed context. We assume some number of systems interconnected by some type of networking facility. Within each system, we assume some function or process within the operating system is responsible for resource allocation. Each such process controls a number of resources and serves a number of user processes. The task is to devise an algorithm by which these processes may cooperate in enforcing mutual exclusion.

Algorithms for mutual exclusion may be either centralized or distributed. In a fully **centralized algorithm**, one node is designated as the control node and controls access to all shared objects. When any process requires access to a critical resource, it issues a Request to its local resource-controlling process. This process, in turn, sends a Request message to the control node, which returns a Reply (permission) message when the shared object becomes available. When a process has finished with a resource, a Release message is sent to the control node. Such a centralized algorithm has two key properties:

1. Only the control node makes resource-allocation decisions.
2. All necessary information is concentrated in the control node, including the identity and location of all resources and the allocation status of each resource.

The centralized approach is straightforward, and it is easy to see how mutual exclusion is enforced: The control node will not satisfy a request for a resource until

that resource has been released. However, such a scheme suffers several drawbacks. If the control node fails, then the mutual exclusion mechanism breaks down, at least temporarily. Furthermore, every resource allocation and deallocation requires an exchange of messages with the control node. Thus, the control node may become a bottleneck.

Because of the problems with centralized algorithms, there has been more interest in the development of distributed algorithms. A fully **distributed algorithm** is characterized by the following properties:

1. All nodes have an equal amount of information, on average.
2. Each node has only a partial picture of the total system and must make decisions based on this information.
3. All nodes bear equal responsibility for the final decision.
4. All nodes expend equal effort, on average, in effecting a final decision.
5. Failure of a node, in general, does not result in a total system collapse.
6. There exists no system-wide common clock with which to regulate the timing of events.

Points 2 and 6 may require some elaboration. With respect to point 2, some distributed algorithms require that all information known to any node be communicated to all other nodes. Even in this case, at any given time, some of that information will be in transit and will not have arrived at all of the other nodes. Thus, because of time delays in message communication, a node's information is usually not completely up to date and is in that sense only partial information.

With respect to point 6, because of the delay in communication among systems, it is impossible to maintain a system-wide clock that is instantly available to all systems. Furthermore, it is also technically impractical to maintain one central clock and to keep all local clocks synchronized precisely to that central clock; over a period of time, there will be some drift among the various local clocks that will cause a loss of synchronization.

It is the delay in communication, coupled with the lack of a common clock, that makes it much more difficult to develop mutual exclusion mechanisms in a distributed system compared to a centralized system. Before looking at some algorithms for distributed mutual exclusion, we examine a common approach to overcoming the clock inconsistency problem.

Ordering of Events in a Distributed System

Fundamental to the operation of most distributed algorithms for mutual exclusion and deadlock is the temporal ordering of events. The lack of a common clock or a means of synchronizing local clocks is thus a major constraint. The problem can be expressed in the following manner. We would like to be able to say that an event a at system i occurred before (or after) event b at system j, and we would like to be able to arrive consistently at this conclusion at all systems in the network. Unfortunately, this statement is not precise for two reasons. First, there may be a delay between the actual occurrence of an event and the time that it is observed on some other system. Second, the lack of synchronization leads to a variance in clock readings on different systems.

To overcome these difficulties, a method referred to as timestamping has been proposed by Lamport [LAMP78], which orders events in a distributed system without using physical clocks. This technique is so efficient and effective that it is used in the great majority of algorithms for distributed mutual exclusion and deadlock.

To begin, we need to decide on a definition of the term *event*. Ultimately, we are concerned with actions that occur at a local system, such as a process entering or leaving its critical section. However, in a distributed system, the way in which processes interact is by means of messages. Therefore, it makes sense to associate events with messages. A local event can be bound to a message very simply; for example, a process can send a message when it desires to enter its critical section or when it is leaving its critical section. To avoid ambiguity, we associate events with the sending of messages only, not with the receipt of messages. Thus, each time that a process transmits a message, an event is defined that corresponds to the time that the message leaves the process.

The timestamping scheme is intended to order events consisting of the transmission of messages. Each system i in the network maintains a local counter, C_i, which functions as a clock. Each time a system transmits a message, it first increments its clock by 1. The message is sent in the form

$$(m, T_i, i)$$

where

m = contents of the message
T_i = timestamp for this message, set to equal C_i
i = numerical identifier of this system in the distributed system

When a message is received, the receiving system j sets its clock to one more than the maximum of its current value and the incoming timestamp:

$$C_j \leftarrow 1 + \max[C_j, T_i]$$

At each site, the ordering of events is determined by the following rules. For a message x from site i and a message y from site j, x is said to precede y if one of the following conditions holds:

1. If $T_i < T_j$, or
2. If $T_i = T_j$ and $i < j$

The time associated with each message is the timestamp accompanying the message, and the ordering of these times is determined by the two foregoing rules. That is, two messages with the same timestamp are ordered by the numbers of their sites. Because the application of these rules is independent of site, this approach avoids any problems of drift among the various clocks of the communicating processes.

An example of the operation of this algorithm is shown in Figure 19.8. There are three sites, each of which is represented by a process that controls the timestamping algorithm. Process P_1 begins with a clock value of 0. To transmit message a, it increments its clock by 1 and transmits $(a, 1, 1)$, where the first numerical value is the timestamp and the second is the identity of the site. This message is received by

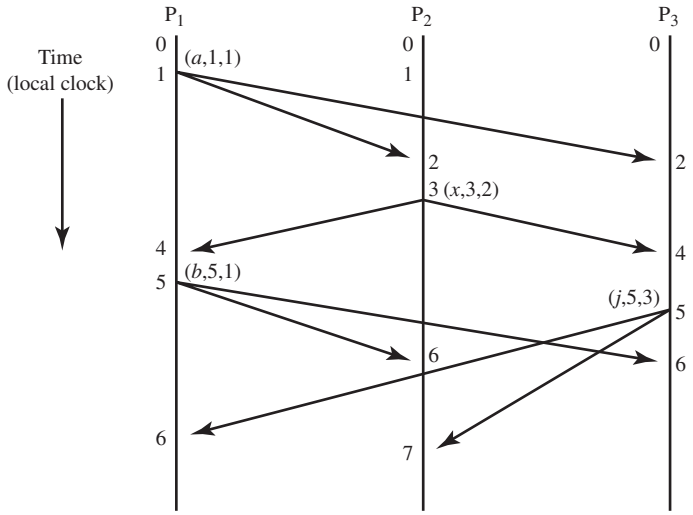


Figure 19.8 Example of Operation of Timestamping Algorithm

processes at sites 2 and 3. In both cases, the local clock has a value of zero and is set to a value of $2 = 1 + \max[0, 1]$. P_2 issues the next message, first incrementing its clock to 3. Upon receipt of this message, P_1 and P_3 increment their clocks to 4. Then P_1 issues message *b* and P_3 issues message *j* at about the same time and with the same timestamp. Because of the ordering principle outlined previously, this causes no confusion. After all of these events have taken place, the ordering of messages is the same at all sites, namely $\{a, x, b, j\}$.

The algorithm works in spite of differences in transmission times between pairs of systems, as illustrated in Figure 19.9. Here, P_1 and P_4 issue messages with the same timestamp. The message from P_1 arrives earlier than that of P_4 at site 2, but later than

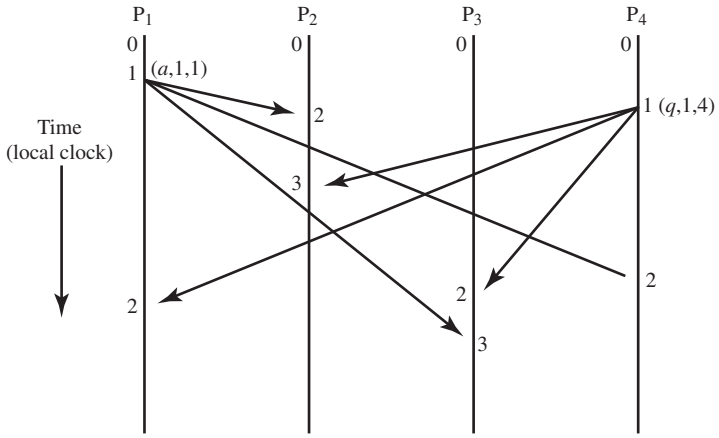


Figure 19.9 Another Example of Operation of Timestamping Algorithm

that of P_4 at site 3. Nevertheless, after all messages have been received at all sites, the ordering of messages is the same at all sites: $\{a, q\}$.

Note the ordering imposed by this scheme does not necessarily correspond to the actual time sequence. For the algorithms based on this timestamping scheme, it is not important which event actually happened first. It is only important that all processes that implement the algorithm agree on the ordering that is imposed on the events.

In the two examples just discussed, each message is sent from one process to all other processes. If some messages are not sent this way, some sites do not receive all of the messages in the system, and it is therefore impossible that all sites have the same ordering of messages. In such a case, a collection of partial orderings exist. However, we are primarily concerned with the use of timestamps in distributed algorithms for mutual exclusion and deadlock detection. In such algorithms, a process usually sends a message (with its timestamp) to every other process, and the timestamps are used to determine how the messages are processed.

Distributed Queue

First Version One of the earliest proposed approaches to providing distributed mutual exclusion is based on the concept of a distributed queue [LAMP78]. The algorithm is based on the following assumptions:

1. A distributed system consists of N nodes, uniquely numbered from 1 to N. Each node contains one process that makes requests for mutually exclusive access to resources on behalf of other processes; this process also serves as an arbitrator to resolve incoming requests from other nodes that overlap in time.
2. Messages sent from one process to another are received in the same order in which they are sent.
3. Every message is correctly delivered to its destination in a finite amount of time.
4. The network is fully connected; this means that every process can send messages directly to every other process, without requiring an intermediate process to forward the message.

Assumptions 2 and 3 can be realized by the use of a reliable transport protocol, such as TCP (Chapter 13).

For simplicity, we describe the algorithm for the case in which each site only controls a single resource. The generalization to multiple resources is trivial.

The algorithm attempts to generalize an algorithm that would work in a straightforward manner in a centralized system. If a single central process managed the resource, it could queue incoming requests and grant requests in a first-in-first-out manner. To achieve this same algorithm in a distributed system, all of the sites must have a copy of the same queue. Timestamping can be used to assure that all sites agree on the order in which resource requests are to be granted. One complication arises: Because it takes some finite amount of time for messages to transit a network, there is a danger that two different sites will not agree on which process is at the head of the queue. Consider Figure 19.9. There is a point at which message a has arrived at P_2

and message q has arrived at P_3, but both messages are still in transit to other processes. Thus, there is a period of time in which P_1 and P_2 consider message a to be the head of the queue and in which P_3 and P_4 consider message q to be the head of the queue. This could lead to a violation of the mutual exclusion requirement. To avoid this, the following rule is imposed: For a process to make an allocation decision based on its own queue, it needs to have received a message from each of the other sites such that the process is guaranteed that no message earlier than its own head of queue is still in transit. This rule is explained in part 3b of the algorithm described subsequently.

At each site, a data structure is maintained that keeps a record of the most recent message received from each site (including the most recent message generated at this site). Lamport refers to this structure as a queue; actually it is an array with one entry for each site. At any instant, entry $q[j]$ in the local array contains a message from P_j. The array is initialized as follows:

$$q[j] = (\text{Release}, 0, j) \quad j = 1, \ldots, N$$

Three types of messages are used in this algorithm:

- (Request, T_i, i): A request for access to a resource is made by P_i.
- (Reply, T_j, j): P_j grants access to a resource under its control.
- (Release, T_k, k): P_k releases a resource previously allocated to it.

The algorithm is as follows:

1. When P_i requires access to a resource, it issues a request (Request, T_i, i), time-stamped with the current local clock value. It puts this message in its own array at $q[i]$ and sends the message to all other processes.
2. When P_j receives (Request, T_i, i), it puts this message in its own array at $q[i]$. If $q[j]$ does not contain a request message, then P_j transmits (Reply, T_j, j) to P_i. It is this action that implements the rule described previously, which assures that no earlier Request message is in transit at the time of a decision.
3. P_i can access a resource (enter its critical section) when both of these conditions hold:
   a. P_i's own Request message in array q is the earliest Request message in the array; because messages are consistently ordered at all sites, this rule permits one and only one process to access the resource at any instant.
   b. All other messages in the local array are later than the message in $q[i]$; this rule guarantees that P_i has learned about all requests that preceded its current request.
3. P_i releases a resource by issuing a release (Release, T_i, i), which it puts in its own array and transmits to all other processes.
4. When P_i receives (Release, T_j, j), it replaces the current contents of $q[j]$ with this message.
5. When P_i receives (Reply, T_j, j), it replaces the current contents of $q[j]$ with this message.

It is easily shown that this algorithm enforces mutual exclusion, is fair, avoids deadlock, and avoids starvation:

- **Mutual exclusion:** Requests for entry into the critical section are handled according to the ordering of messages imposed by the timestamping mechanism. Once P_i decides to enter its critical section, there can be no other Request message in the system that was transmitted before its own. This is true because P_i has by then necessarily received a message from all other sites and these messages from other sites date from later than its own Request message. We can be sure of this because of the Reply message mechanism; remember that messages between two sites cannot arrive out of order.
- **Fair:** Requests are granted strictly on the basis of timestamp ordering. Therefore, all processes have equal opportunity.
- **Deadlock free:** Because the timestamp ordering is consistently maintained at all sites, deadlock cannot occur.
- **Starvation free:** Once P_i has completed its critical section, it transmits the Release message. This has the effect of deleting P_i's Request message at all other sites, allowing some other process to enter its critical section.

As a measure of efficiency of this algorithm, note to guarantee exclusion, $3 \times (N - 1)$ messages are required: $(N - 1)$ Request messages, $(N - 1)$ Reply messages, and $(N - 1)$ Release messages.

Second Version A refinement of the Lamport algorithm was proposed in [RICA81]. It seeks to optimize the original algorithm by eliminating Release messages. The same assumptions as before are in force, except that it is not necessary that messages sent from one process to another are received in the same order in which they are sent.

As before, each site includes one process that controls resource allocation. This process maintains an array q and obeys the following rules:

1. When P_i requires access to a resource, it issues a request (Request, T_i, i), timestamped with the current local clock value. It puts this message in its own array at $q[i]$ and sends the message to all other processes.
2. When P_j receives (Request, T_i, i), it obeys the following rules:
   - **a.** If P_j is currently in its critical section, it defers sending a Reply message (see Rule 4, which follows)
   - **b.** If P_j is not waiting to enter its critical section (has not issued a Request that is still outstanding), it transmits (Reply, T_j, j) to P_i.
   - **c.** If P_j is waiting to enter its critical section and if the incoming message follows P_j's request, then it puts this message in its own array at $q[i]$ and defers sending a Reply message.
   - **d.** If P_j is waiting to enter its critical section and if the incoming message precedes P_j's request, then it puts this message in its own array at $q[i]$ and transmits (Reply, T_j, j) to P_i.

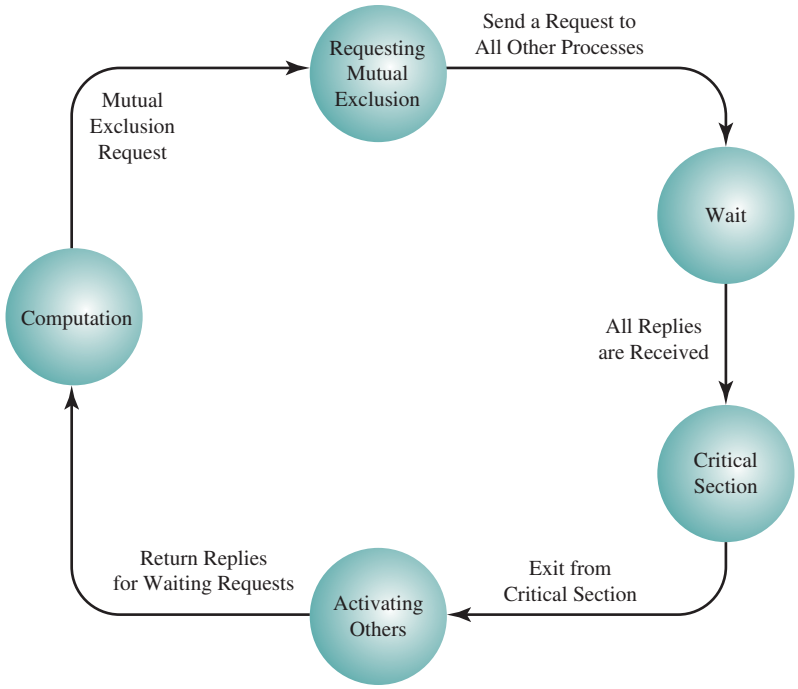


Figure 19.10 State Diagram for Algorithm in [RICA81]

5. P_i can access a resource (enter its critical section) when it has received a Reply message from all other processes.
6. When P_i leaves its critical section, it releases the resource by sending a Reply message to each pending Request.

The state transition diagram for each process is shown in Figure 19.10.

To summarize, when a process wishes to enter its critical section, it sends a time-stamped Request message to all other processes. When it receives a Reply from all other processes, it may enter its critical section. When a process receives a Request from another process, it must eventually send a matching Reply. If a process does not wish to enter its critical section, it sends a Reply at once. If it wants to enter its critical section, it compares the timestamp of its Request with that of the last Request received, and if the latter is more recent, it defers its Reply; otherwise Reply is sent at once.

With this method, $2 \times (N - 1)$ messages are required: $(N - 1)$ Request messages to indicate P_i's intention of entering its critical section, and $(N - 1)$ Reply messages to allow the access it has requested.

The use of timestamping in this algorithm enforces mutual exclusion. It also avoids deadlock. To prove the latter, assume the opposite: It is possible that, when there are no more messages in transit, we have a situation in which each process has transmitted a Request and has not received the necessary Reply. This situation cannot arise, because a decision to defer a Reply is based on a relation that orders Requests. There is therefore one Request that has the earliest timestamp and that will receive all the necessary Replies. Deadlock is therefore impossible.

Starvation is also avoided because Requests are ordered. Because Requests are served in that order, every Request will at some stage become the oldest and will then be served.

A Token-Passing Approach

A number of investigators have proposed a quite different approach to mutual exclusion, which involves passing a token among the participating processes. The token is an entity that at any time is held by one process. The process holding the token may enter its critical section without asking permission. When a process leaves its critical section, it passes the token to another process.

In this subsection, we look at one of the most efficient of these schemes. It was first proposed in [SUZU82]; a logically equivalent proposal also appeared in [RICA83]. For this algorithm, two data structures are needed. The token, which is passed from process to process, is actually an array, `token`, whose kth element records the timestamp of the last time that the token visited process P_k. In addition, each process maintains an array, request, whose jth element records the timestamp of the last Request received from P_j.

The procedure is as follows. Initially, the token is assigned arbitrarily to one of the processes. When a process wishes to use its critical section, it may do so if it currently possesses the token; otherwise it broadcasts a timestamped request message to all other processes and waits until it receives the token. When process P_j leaves its critical section, it must transmit the token to some other process. It chooses the next process to receive the token by searching the request array in the order $j + 1, j + 2, \ldots, 1, 2, \ldots, j - 1$ for the first entry request $[k]$ such that the timestamp for P_k's last request for the token is greater than the value recorded in the token for P_k's last holding of the token, that is, request $[k] >$ token $[k]$.

Figure 19.11 depicts the algorithm, which is in two parts. The first part deals with the use of the critical section and consists of a prelude, followed by the critical section, followed by a postlude. The second part concerns the action to be taken upon receipt of a request. The variable clock is the local counter used for the timestamp function. The operation wait (access, token) causes the process to wait until a message of the type "access" is received, which is then put into the variable array token.

The algorithm requires either of the following:

- N messages ($N - 1$ to broadcast the request and 1 to transfer the token) when the requesting process does not hold the token
- No messages, if the process already holds the token

19.4 DISTRIBUTED DEADLOCK

In Chapter 6, we defined deadlock as the permanent blocking of a set of processes that either compete for system resources or communicate with one another. This definition is valid for a single system as well as for a distributed system. As with mutual exclusion, deadlock presents more complex problems in a distributed system, compared with a shared memory system. Deadlock handling is complicated in a distributed system

```
if  (!token_present) {
  clock++;                /* Prelude */
  broadcast (Request, clock, i);
  wait (access, token);
  token_present = true;
}

token_held = true;
<critical section>;

token[i] = clock;         /* Postlude */
token_held = false;
 for  (int  j = i + 1; j < n; j++) {
  if  (request(j) > token[j] && token_present) {
    token_present = false;
    send (access, token[j]);
  }
}
```

(a) First Part

```
if  (received (Request, k, j)) {
  request (j) = max(request(j), k);
  if  (token_present && !token_held)
    <text of postlude>;
}
```

(b) Second Part

Notation	
send (j, access, token)	end message of type access, with token, by process j
broadcast (request, clock, i)	send message from process i of type request, with timestamp clock, to all other processes
received (request, t, j)	receive message from process j of type request, with timestamp t

Figure 19.11 Token-Passing Algorithm (for process P_i)

because no node has accurate knowledge of the current state of the overall system and because every message transfer between processes involves an unpredictable delay.

Two types of distributed deadlock have received attention in the literature: those that arise in the allocation of resources, and those that arise with the communication of messages. In resource deadlocks, processes attempt to access resources, such as data objects in a database or I/O resources on a server; deadlock occurs if each process in a set of processes requests a resource held by another process in the set. In communications deadlocks, messages are the resources for which processes wait; deadlock occurs if each process in a set is waiting for a message from another process in the set, and no process in the set ever sends a message.

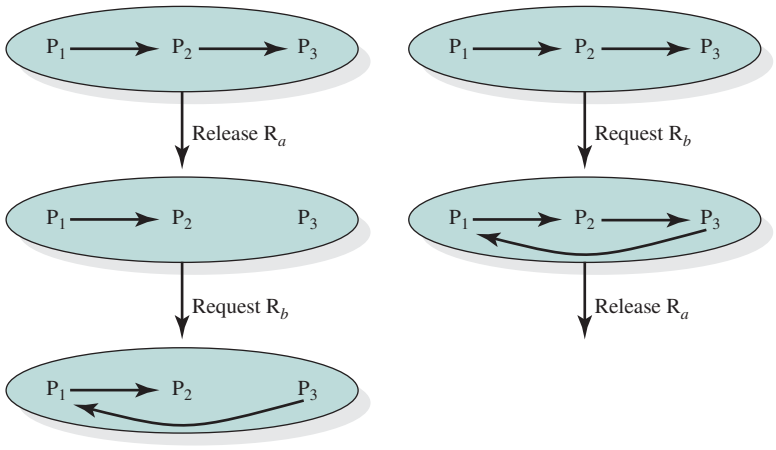
Deadlock in Resource Allocation

Recall from Chapter 6 that a deadlock in resource allocation exists only if all of the following conditions are met:

- **Mutual exclusion:** Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.
- **Hold and wait:** A process may hold allocated resources while awaiting assignment of others.
- **No preemption:** No resource can be forcibly removed from a process holding it.
- **Circular wait:** A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

The aim of an algorithm that deals with deadlock is either to prevent the formation of a circular wait, or to detect its actual or potential occurrence. In a distributed system, the resources are distributed over various sites and access to them is regulated by control processes that do not have complete, up-to-date knowledge of the global state of the system and must therefore make their decisions on the basis of local information. Thus, new deadlock algorithms are required.

One example of the difficulty faced in distributed deadlock management is the phenomenon of phantom deadlock. An example of phantom deadlock is illustrated in Figure 19.12. The notation $P_1 \rightarrow P_2 \rightarrow P_3$ means that P_1 is halted waiting for a resource held by P_2, and P_2 is halted waiting for a resource held by P_3. Let us say that at the beginning of the example, P_3 owns resource R_a and P_1 owns resource R_b. Suppose now that P_3 issues first a message releasing R_a then a message requesting R_b. If the first message reaches a cycle-detecting process before the second, the sequence of Figure 19.12a results, which properly reflects resource requirements. If, however, the second message arrives before the first message, a deadlock is registered (see Figure 19.12b). This is a false detection, not a real deadlock, due to the lack of a global state, such as would exist in a centralized system.



(a) Release arrives before request (b) Request arrives before release

Figure 19.12 Phantom Deadlock

DEADLOCK PREVENTION Two of the deadlock prevention techniques discussed in Chapter 6 can be used in a distributed environment.

1. The circular-wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering. A major disadvantage of this method is that resources may not be requested in the order in which they are used; thus, resources may be held longer than necessary.
2. The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time, and blocking the process until all requests can be granted simultaneously. This approach is inefficient in two ways. First, a process may be held up for a long time waiting for all of its resource requests to be filled, when in fact it could have proceeded with only some of the resources. Second, resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes.

Both of these methods require that a process determine its resource requirements in advance. This is not always the case; an example is a database application in which new items can be added dynamically. As an example of an approach that does not require this foreknowledge, we consider two algorithms proposed in [ROSE78]. These were developed in the context of database work, so we shall speak of transactions rather than processes.

The proposed methods make use of timestamps. Each transaction carries throughout its lifetime the timestamp of its creation. This establishes a strict ordering of the transactions. If a resource R already being used by transaction T1 is requested by another transaction T2, the conflict is resolved by comparing their timestamps. This comparison is used to prevent the formation of a circular-wait condition. Two variations of this basic method are proposed by the authors, referred to as the "wait-die" method and the "wound-wait" method.

Let us suppose T1 currently holds R and T2 issues a request. For the **wait-die method**, Figure 19.13a shows the algorithm used by the resource allocator at the site of R. The timestamps of the two transactions are denoted as e(T1) and e(T2). If T2 is older, it is blocked until T1 releases R, either by actively issuing a release or by being "killed" when requesting another resource. If T2 is younger, then T2 is restarted but with the same timestamp as before.

Thus, in a conflict, the older transaction takes priority. Because a killed transaction is revived with its original timestamp, it grows older and therefore gains increased priority. No site needs to know the state of allocation of all resources. All that are required are the timestamps of the transactions that request its resources.

```
if (e(T2) < e(T1))
   halt_T2 ('wait');
else
   kill_T2 ('die');
```

(a) Wait-die method

```
if (e(T2) < e(T1))
   kill_T1 ('wound');
else
   halt_T2 ('wait');
```

(b) Wound-wait method

Figure 19.13 Deadlock Prevention Methods

The **wound-wait method** immediately grants the request of an older transaction by killing a younger transaction that is using the required resource. This is shown in Figure 19.13b. In contrast to the wait-die method, a transaction never has to wait for a resource being used by a younger transaction.

Deadlock Avoidance Deadlock avoidance is a technique in which a decision is made dynamically whether a given resource allocation request could, if granted, lead to a deadlock. [SING94] points out that distributed deadlock avoidance is impractical for the following reasons:

1. Every node must keep track of the global state of the system; this requires substantial storage and communications overhead.
2. The process of checking for a safe global state must be mutually exclusive. Otherwise, two nodes could each be considering the resource request of a different process and concurrently reach the conclusion that it is safe to honor the request, when in fact if both requests are honored, deadlock will result.
3. Checking for safe states involves considerable processing overhead for a distributed system with a large number of processes and resources.

Deadlock Detection With deadlock detection, processes are allowed to obtain free resources as they wish, and the existence of a deadlock is determined after the fact. If a deadlock is detected, one of the *constituent* processes is selected and required to release the resources necessary to break the deadlock.

The difficulty with distributed deadlock detection is that each site only knows about its own resources, whereas a deadlock may involve distributed resources. Several approaches are possible, depending on whether the system control is centralized, hierarchical, or distributed (see Table 19.1).

With **centralized control**, one site is responsible for deadlock detection. All request and release messages are sent to the central process as well as to the process that controls the particular resource. Because the central process has a complete picture, it is in a position to detect a deadlock. This approach requires a lot of messages and is vulnerable to a failure of the central site. In addition, phantom deadlocks may be detected.

With **hierarchical control**, the sites are organized in a tree structure, with one site serving as the root of the tree. At each node, other than leaf nodes, information about the resource allocation of all dependent nodes is collected. This permits deadlock detection to be done at lower levels than the root node. Specifically, a deadlock that involves a set of resources will be detected by the node that is the common ancestor of all sites whose resources are among the objects in conflict.

With **distributed control**, all processes cooperate in the deadlock detection function. In general, this means that considerable information must be exchanged, with timestamps; thus the overhead is significant. [RAYN88] cites a number of approaches based on distributed control, and [DATT90] provides a detailed examination of one approach.

We now give an example of a distributed deadlock detection algorithm ([DATT92], [JOHN91]). The algorithm deals with a distributed database system in which each site maintains a portion of the database and transactions may be initiated from each site. A transaction can have at most one outstanding resource request. If

Table 19.1 Distributed Deadlock Detection Strategies

Centralized Algorithms		Hierarchical Algorithms		Distributed Algorithms	
Strengths	**Weaknesses**	**Strengths**	**Weaknesses**	**Strengths**	**Weaknesses**
• Algorithms are conceptually simple and easy to implement. • Central site has complete information and can optimally resolve deadlocks.	• Considerable communications overhead; every node must send state information to central node. • Vulnerable to failure of central node.	• Not vulnerable to single point of failure. • Deadlock resolution activity is limited if most potential deadlocks are relatively localized.	• May be difficult to configure system so that most potential deadlocks are localized; otherwise there may actually be more overhead than in a distributed approach.	• Not vulnerable to single point of failure. • No node is swamped with deadlock detection activity.	• Deadlock resolution is cumbersome because several sites may detect the same deadlock and may not be aware of other nodes involved in the deadlock. • Algorithms are difficult to design because of timing considerations.

a transaction needs more than one data object, the second data object can be requested only after the first data object has been granted.

Associated with each data object *i* at a site are two parameters: a unique identifier D_i and the variable Locked_by (D_i). This latter variable has the value nil if the data object is not locked by any transaction; otherwise its value is the identifier of the locking transaction.

Associated with each transaction *j* at a site are four parameters:

- A unique identifier T_j
- The variable Held_by (T_j), which is set to nil if transaction T_j is executing or in a Ready state. Otherwise, its value is the transaction that is holding the data object required by transaction T_j.
- The variable Wait_for (T_j), which has the value nil if transaction T_i is not waiting for any other transaction. Otherwise, its value is the identifier of the transaction that is at the head of an ordered list of transactions that are blocked.
- A queue Request_Q (T_j), which contains all outstanding requests for data objects being held by T_j. Each element in the queue is of the form (T_k, D_k), where T_k is the requesting transaction and D_k is the data object held by T_j.

For example, suppose transaction T_2 is waiting for a data object held by T_1, which is, in turn, waiting for a data object held by T_0. Then the relevant parameters have the following values:

Transaction	Wait_for	Held_by	Request_Q
T_0	nil	nil	T_1
T_1	T_0	T_0	T_2
T_2	T_0	T_1	nil

This example highlights the difference between Wait_for (T_i) and Held_by (T_i). Neither process can proceed until T_0 releases the data object needed by T_1, which can then execute and release the data object needed by T_2.

Figure 19.14 shows the algorithm used for deadlock detection. When a transaction makes a lock request for a data object, a server process associated with that data object either grants or denies the request. If the request is not granted, the server process returns the identity of the transaction holding the data object.

When the requesting transaction receives a granted response, it locks the data object. Otherwise, the requesting transaction updates its Held_by variable to the identity of the transaction holding the data object. It adds its identity to the Request_Q of the holding transaction. It updates its Wait_for variable either to the identity of the holding transaction (if that transaction is not waiting) or to the identity of the Wait_for variable of the holding transaction. In this way, the Wait_for variable is set to the value of the transaction that ultimately is blocking execution. Finally, the requesting transaction issues an update message to all of the transactions in its own Request_Q to modify all the Wait_for variables that are affected by this change.

When a transaction receives an update message, it updates its Wait_for variable to reflect the fact that the transaction on which it had been ultimately waiting is now blocked by yet another transaction. Then it does the actual work of deadlock detection by checking to see if it is now waiting for one of the processes that is waiting for it. If not, it forwards the update message. If so, the transaction sends a clear message to the transaction holding its requested data object and allocates every data object that it holds to the first requester in its Request_Q and enqueues remaining requesters to the new transaction.

An example of the operation of the algorithm is shown in Figure 19.15. When T_0 makes a request for a data object held by T_3, a cycle is created. T_0 issues an update message that propagates from T_1 to T_2 to T_3. At this point, T_3 discovers that the intersection of its Wait_for and Request_Q variables is not empty. T_3 sends a clear message to T_2 so T_3 is purged from Request_Q (T_2), and it releases the data objects it held, activating T_4 and T_6.

Deadlock in Message Communication

Mutual Waiting Deadlock occurs in message communication when each of a group of processes is waiting for a message from another member of the group and there are no messages in transit.

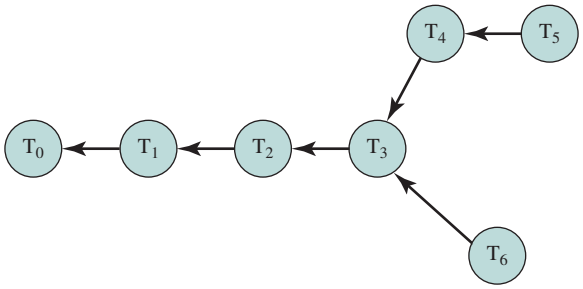
To analyze this situation in more detail, we define the dependence set (DS) of a process. For a process P_i that is halted, waiting for a message, DS(P_i) consists of all processes from which P_i is expecting a message. Typically, P_i can proceed if any of the expected messages arrives. An alternative formulation is that P_i can proceed only after all of the expected messages arrive. The former situation is the more common one and is considered here.

With the preceding definition, a deadlock in a set S of processes can be defined as follows:

1. All the processes in S are halted, waiting for messages.
2. S contains the dependence set of all processes in S.
3. No messages are in transit between members of S.

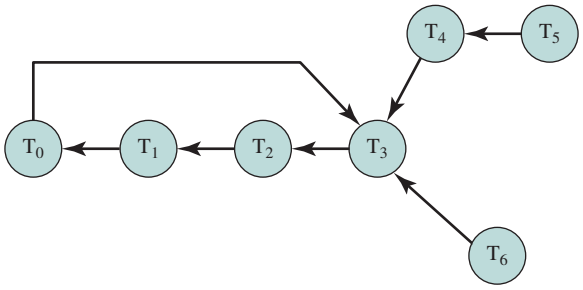
```
/* Data object Dj receiving a lock_request(Ti) */
  if (Locked_by(Dj) == null)
    send(granted);
  else {
    send not granted to Ti;
    send Locked_by(Dj) to Ti
  }
/* Transaction Ti makes a lock request for data object Dj  */
  send lock_request(Ti) to Dj;
  wait for granted/not granted;
  if (granted) {
    Locked_by(Dj) = Ti;
    Held_by(Ti) = f;
  }
  else { /* suppose Dj is being used by transaction Tj  */
    Held_by(Ti) = Tj;
    Enqueue(Ti, Request_Q(Tj));
    if (Wait_for(Tj) == null)
      Wait_for(Ti) = Tj  ;
    else
      Wait_for(Ti) = Wait_for(Tj);
      update(Wait_for(Ti), Request_Q(Ti));
  }
/* Transaction Tj receiving an update message */
  if (Wait_for(Tj) != Wait_for(Ti))
    Wait_for(Tj) = Wait_for(Ti);
  if (intersect(Wait_for(Tj), Request_Q(Tj)) = null)
    update(Wait_for(Ti), Request_Q(Tj);
  else  {
    DECLARE DEADLOCK;
      /* initiate deadlock resolution as follows */
      /* Tj is chosen as the transaction to be aborted */
      /* Tj releases all the data objects it holds */
      send_clear(Tj, Held_by(Tj));
      allocate each data object Di held by Tj to the first requester Tk
      in Request_Q(Tj);
      for (every transaction Tn  in Request_Q(Tj) requesting data object
          Di  held by Tj)
      {
        Enqueue(Tn, Request_Q(Tk));
      }
  }
/* Transaction Tk  receiving a clear(Tj, Tk) message */
    purge the tuple having Tj as the requesting transaction from
    Request_Q(Tk);
```

Figure 19.14 **A Distributed Deadlock Detection Algorithm**



Transaction	Wait_for	Held_by	Request_Q
T_0	nil	nil	T_1
T_1	T_0	T_0	T_2
T_2	T_0	T_1	T_3
T_3	T_0	T_2	T_4, T_6
T_4	T_0	T_3	T_5
T_5	T_0	T_4	nil
T_6	T_0	T_3	nil

(a) State of system before request



Transaction	Wait_for	Held_by	Request_Q
T_0	T_0	T_3	T_1
T_1	T_0	T_0	T_2
T_2	T_0	T_1	T_3
T_3	T_0	T_2	T_4, T_6, T_0
T_4	T_0	T_3	T_5
T_5	T_0	T_4	NIL
T_6	T_0	T_3	NIL

(b) State of system after T_0 makes a request to T_3

Figure 19.15 Example of Distributed Deadlock Detection Algorithm of Figure 19.14

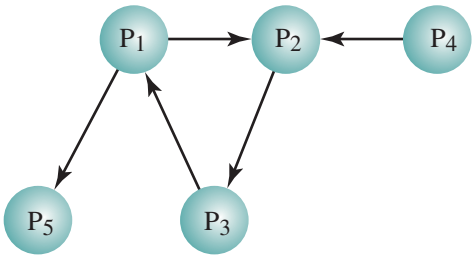
Any process in S is deadlocked because it can never receive a message that will release it.

In graphical terms, there is a difference between message deadlock and resource deadlock. With resource deadlock, a deadlock exists if there is a closed loop, or cycle, in the graph that depicts process dependencies. In the resource case, one process is dependent on another if the latter holds a resource that the former requires. With message deadlock, the condition for deadlock is that all successors of any member of S are themselves in S.

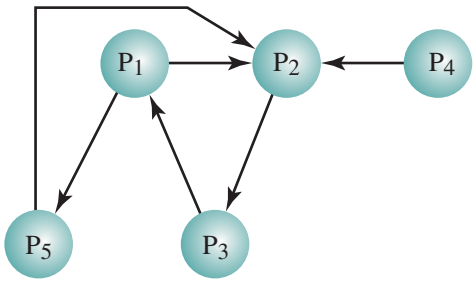
Figure 19.16 illustrates the point. In Figure 19.16a, P_1 is waiting for a message from either P_2 or P_5; P_5 is not waiting for any message and so can send a message to P_1, which is therefore released. As a result, the links (P_1, P_5) and (P_1, P_2) are deleted. Figure 19.16b adds a dependency: P_5 is waiting for a message from P_2, which is waiting for a message from P_3, which is waiting for a message from P_1, which is waiting for a message from P_2. Thus, deadlock exists.

As with resource deadlock, message deadlock can be attacked by either prevention or detection. [RAYN88] gives some examples.

Unavailability of Message Buffers Another way in which deadlock can occur in a message-passing system has to do with the allocation of buffers for the storage of messages in transit. This kind of deadlock is well known in packet-switching data networks. We first examine this problem in the context of a data network, then view it from the point of view of a distributed operating system.



(a) No deadlock



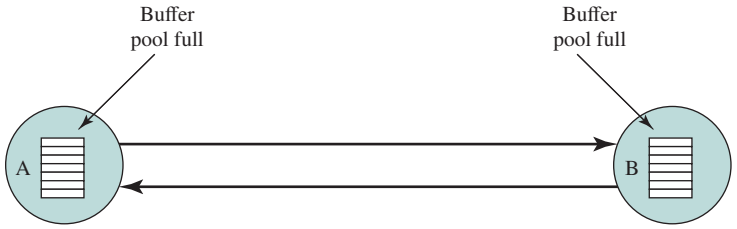
(b) Deadlock

Figure 19.16 Deadlock in Message Communication

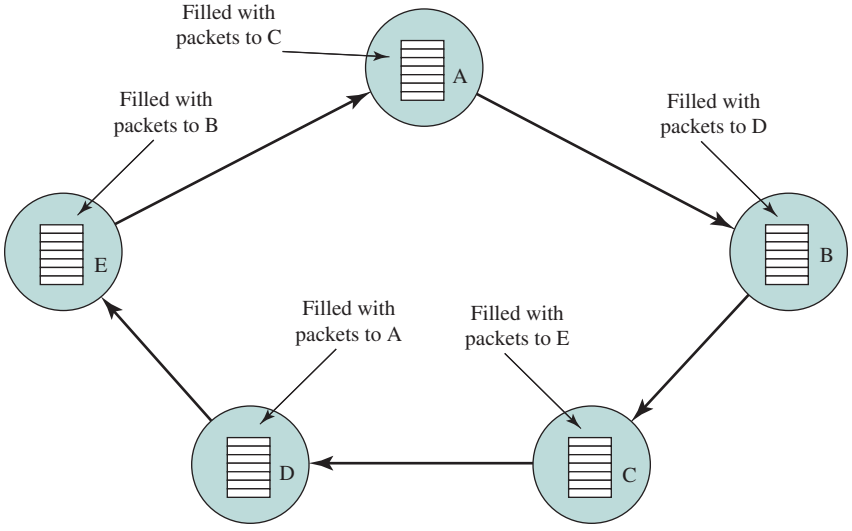
The simplest form of deadlock in a data network is direct store-and-forward deadlock and can occur if a packet-switching node uses a common buffer pool from which buffers are assigned to packets on demand. Figure 19.17a shows a situation in which all of the buffer space in node A is occupied with packets destined for B. The reverse is true at B. Neither node can accept any more packets because their buffers are full. Thus neither node can transmit or receive on any link.

Direct store-and-forward deadlock can be prevented by not allowing all buffers to end up dedicated to a single link. Using separate fixed-size buffers, one for each link, will achieve this prevention. Even if a common buffer pool is used, deadlock is avoided if no single link is allowed to acquire all of the buffer space.

A more subtle form of deadlock, indirect store-and-forward deadlock, is illustrated in Figure 19.17b. For each node, the queue to the adjacent node in one direction is full with packets destined for the next node beyond. One simple way to prevent this type of deadlock is to employ a structured buffer pool (see Figure 19.18). The buffers are organized in a hierarchical fashion. The pool of memory at level 0 is



(a) Direct store-and-forward deadlock



(b) Indirect store-and-forward deadlock

Figure 19.17 Store-and-Forward Deadlock

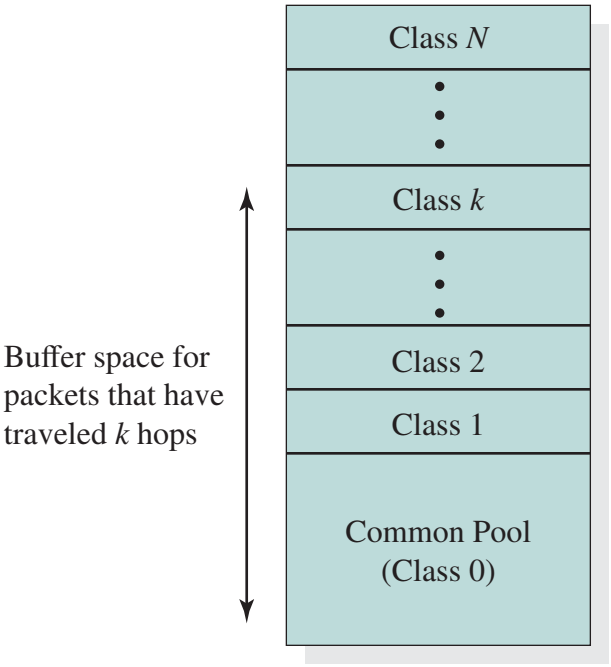


Figure 19.18 Structured Buffer Pool for Deadlock Prevention

unrestricted; any incoming packet can be stored there. From level 1 to level N (where N is the maximum number of hops on any network path), buffers are reserved in the following way: Buffers at level k are reserved for packets that have traveled at least k hops so far. Thus, in heavy load conditions, buffers fill up progressively from level 0 to level N. If all buffers up through level k are filled, arriving packets that have covered k or less hops are discarded. It can be shown [GOPA85] that this strategy eliminates both direct and indirect store-and-forward deadlocks.

The deadlock problem just described would be dealt with in the context of communications architecture, typically at the network layer. The same sort of problem can arise in a distributed operating system that uses message passing for interprocess communication. Specifically, if the send operation is nonblocking, then a buffer is required to hold outgoing messages. We can think of the buffer used to hold messages to be sent from process X to process Y to be a communications channel between X and Y. If this channel has finite capacity (finite buffer size), then it is possible for the send operation to result in process suspension. That is, if the buffer is of size n and there are currently n messages in transit (not yet received by the destination process), then the execution of an additional send will block the sending process until a receive has opened up space in the buffer.

Figure 19.19 illustrates how the use of finite channels can lead to deadlock. The figure shows two channels, each with a capacity of four messages, one from process X to process Y and one from Y to X. If exactly four messages are in transit in each of the channels, and both X and Y attempt a further transmission before executing a receive, then both are suspended and a deadlock arises.

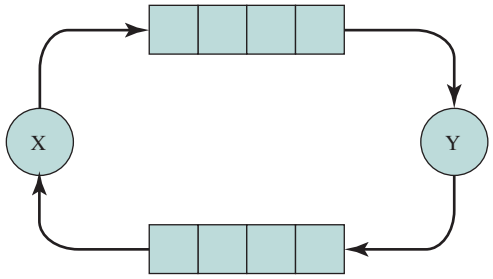


Figure 19.19 **Communication Deadlock in a Distributed System**

If it is possible to establish upper bounds on the number of messages that will ever be in transit between each pair of processes in the system, then the obvious prevention strategy would be to allocate as many buffer slots as needed for all these channels. This might be extremely wasteful, and of course requires this foreknowledge. If requirements cannot be known ahead of time, or if allocating based on upper bounds is deemed too wasteful, then some estimation technique is needed to optimize the allocation. It can be shown that this problem is unsolvable in the general case; some heuristic strategies for coping with this situation are suggested in [BARB90].

19.5 SUMMARY

A distributed operating system may support process migration. This is the transfer of a sufficient amount of the state of a process from one machine to another for the process to execute on the target machine. Process migration may be used for load balancing, to improve performance by minimizing communication activity, to increase availability, or to allow processes access to specialized remote facilities.

With a distributed system, it is often important to establish global state information, to resolve contention for resources, and to coordinate processes. Because of the variable and unpredictable time delay in message transmission, care must be taken to assure that different processes agree on the order in which events have occurred.

Process management in a distributed system includes facilities for enforcing mutual exclusion and for taking action to deal with deadlock. In both cases, the problems are more complex than those in a single system.

19.6 REFERENCES

ANDR90 Andrianoff, S. "A Module on Distributed Systems for the Operating System Course." *Proceedings, Twenty-First SIGCSE Technical Symposium on Computer Science Education, SIGSCE Bulletin,* February 1990.

ARTS89a Artsy, Y., ed. "Special Issue on Process Migration." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems,* Winter 1989.

ARTS89b Artsy, Y. "Designing a Process Migration Facility: The Charlotte Experience." *Computer,* September 1989.

BARB90 Barbosa, V. "Strategies for the Prevention of Communication Deadlocks in Distributed Parallel Programs." *IEEE Transactions on Software Engineering,* November 1990.

BEN06 Ben-Ari, M. *Principles of Concurrent and Distributed Programming*. Harlow, England: Addison-Wesley, 2006.

CABR86 Cabrear, L. "The Influence of Workload on Load Balancing Strategies." *USENIX Conference Proceedings,* Summer 1986.

CASA94 Casavant, T., and Singhal, M. *Distributed Computing Systems.* Los Alamitos, CA: IEEE Computer Society Press, 1994.

CHAN90 Chandras, R. "Distributed Message Passing Operating Systems." *Operating Systems Review*, January 1990.

DATT90 Datta, A., and Ghosh, S. "Deadlock Detection in Distributed Systems." *Proceedings, Phoenix Conference on Computers and Communications,* March 1990.

DATT92 Datta, A.; Javagal, R.; and Ghosh, S. "An Algorithm for Resource Deadlock Detection in Distributed Systems," *Computer Systems Science and Engineering*, October 1992.

DOUG89 Douglas, F., and Ousterhout, J. "Process Migration in Sprite: A Status Report." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems,* Winter 1989.

DOUG91 Douglas, F., and Ousterhout, J. "Transparent Process Migration: Design Alternatives and the Sprite Implementation." *Software Practice and Experience*, August 1991.

EAGE86 Eager, D.; Lazowska, E.; and Zahnorjan, J. "Adaptive Load Sharing in Homogeneous Distributed Systems." *IEEE Transactions on Software Engineering,* May 1986.

ESKI90 Eskicioglu, M. "Design Issues of Process Migration Facilities in Distributed Systems." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments*, Summer 1990.

FINK89 Finkel, R. "The Process Migration Mechanism of Charlotte." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems,* Winter 1989.

GOPA85 Gopal, I. "Prevention of Store-and-Forward Deadlock in Computer Networks." *IEEE Transactions on Communications,* December 1985.

JOHN91 Johnston, B.; Javagal, R.; Datta, A.; and Ghosh, S. "A Distributed Algorithm for Resource Deadlock Detection." *Proceedings, Tenth Annual Phoenix Conference on Computers and Communications*, March 1991.

JUL88 Jul, E.; Levy, H.; Hutchinson, N.; and Black, A. "Fine-Grained Mobility in the Emerald System." *ACM Transactions on Computer Systems,* February 1988.

JUL89 Jul, E. "Migration of Light-Weight Processes in Emerald." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.

LAMP78 Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM,* July 1978.

LELA86 Leland, W., and Ott, T. "Load-Balancing Heuristics and Process Behavior." *Proceedings, ACM SigMetrics Performance 1986 Conference*, 1986.

LYNC96 Lynch, N. *Distributed Algorithms.* San Francisco, CA: Morgan Kaufmann, 1996.

MILO00 Milojicic, D.; Douglis, F.; Paindaveine, Y.; Wheeler, R.; and Zhou, S. "Process Migration." *ACM Computing Surveys*, September 2000.

POPE85 Popek, G., and Walker, B. *The LOCUS Distributed System Architecture*, Cambridge, MA: MIT Press, 1985.

RAYN88 Raynal, M. *Distributed Algorithms and Protocols.* New York: Wiley, 1988.

RICA81 Ricart, G., and Agrawala, A. "An Optimal Algorithm for Mutual Exclusion in Computer Networks." *Communications of the ACM,* January 1981 (Corrigendum in *Communications of the ACM,* September 1981).

RICA83 Ricart, G., and Agrawala, A. "Author's Response to 'On Mutual Exclusion in Computer Networks' by Carvalho and Roucairol." *Communications of the ACM,* February 1983.

ROSE78 Rosenkrantz, D.; Stearns, R.; and Lewis, P. "System Level Concurrency Control in Distributed Database Systems." *ACM Transactions on Database Systems,* June 1978.

SHIV92 Shivaratri, N.; Krueger, P.; and Singhal, M. "Load Distributing for Locally Distributed Systems." *Computer*, December 1992.

SING94 Singhal, M. "Deadlock Detection in Distributed Systems." In [CASA94].

SMIT88 Smith, J. "A Survey of Process Migration Mechanisms." *Operating Systems Review*, July 1988.

SMIT89 Smith, J. "Implementing Remote *fork()* with Checkpoint/restart." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems,* Winter 1989.

SUZU82 Suzuki, I., and Kasami, T. "An Optimality Theory for Mutual Exclusion Algorithms in Computer Networks." *Proceedings of the Third International Conference on Distributed Computing Systems,* October 1982.

WALK89 Walker, B., and Mathews, R. "Process Migration in AIX's Transparent Computing Facility." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems,* Winter 1989.

ZAJC93 Zajcew, R., et al. "An OSF/1 UNIX for Massively Parallel Multicomputers." *Proceedings, Winter USENIX Conference*, January 1993.

19.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

channel	eviction	preemptive transfer
distributed deadlock	global state	process migration
distributed mutual exclusion	nonpreemptive transfer	snapshot

Review Questions

19.1. Discuss some of the reasons for implementing process migration.

19.2. How is the process address space handled during process migration?

19.3. What are the motivations for preemptive and nonpreemptive process migration?

19.4. Why is it impossible to determine a true global state?

19.5. What is the difference between distributed mutual exclusion enforced by a centralized algorithm and enforced by a distributed algorithm?

19.6. Define the two types of distributed deadlock.

Problems

19.1. The flushing policy is described in the subsection on process migration strategies in Section 19.1.

a. From the perspective of the source, which other strategy does flushing resemble?

b. From the perspective of the target, which other strategy does flushing resemble?

19.2. For Figure 19.9, it is claimed that all four processes assign an ordering of $\{a, q\}$ to the two messages, even though q arrives before a at P_3. Work through the algorithm to demonstrate the truth of the claim.

19.3. For Lamport's algorithm, are there any circumstances under which P_i can save itself the transmission of a Reply message?

19.4. For the mutual exclusion algorithm of [RICA81],

a. Prove that mutual exclusion is enforced.

b. If messages do not arrive in the order that they are sent, the algorithm does not guarantee that critical sections are executed in the order of their requests. Is starvation possible?

19.5. In the token-passing mutual exclusion algorithm, is the timestamping used to reset clocks and correct drifts, as in the distributed queue algorithms? If not, what is the function of the timestamping?

19.6. For the token-passing mutual exclusion algorithm, prove that it:

a. guarantees mutual exclusion.

b. avoids deadlock.

c. is fair.

19.7. In Figure 19.11b, explain why the second line cannot simply read "request $(j) = t$."

CHAPTER 20

Overview of Probability and Stochastic Processes

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Understand the basic concepts of probability.
- Explain the concept of random variable.
- Understand some of the important basic concepts of stochastic processes.

Before setting out on our exploration of queueing analysis, we review background on probability and stochastic processes. The reader familiar with these topics can safely skip this chapter.

The chapter begins with an introduction to some elementary concepts from probability theory and random variables; this material is needed for Chapter 21, on queueing analysis. Following this, we look at stochastic processes, which are also relevant to queueing analysis.

20.1 PROBABILITY

We give here the barest outline of probability theory, but enough to support the rest of this chapter.

Definitions of Probability

Probability is concerned with the assignment of numbers to events. The probability $\Pr[A]$ of an event A is a number between 0 and 1 that corresponds to the likelihood that the event A will occur. Generally, we talk of performing an experiment and obtaining an **outcome**. The **event** A is a particular outcome or set of outcomes, and a probability is assigned to that event.

It is difficult to get a firm grip on the concept of probability. Different applications of the theory present probability in different lights. In fact, there are a number of different definitions of probability. We highlight three here.

AXIOMATIC DEFINITION A formal approach to probability is to state a number of axioms that define a probability measure and, from them, to derive laws of probability that can be used to perform useful calculations. The axioms are simply assertions that must be accepted. Once the axioms are accepted, it is possible to prove each of the laws.

The axioms and laws make use of the following concepts from set theory. The **certain event** Ω is the event that occurs in every experiment; it consists of the universe, or **sample space**, of all possible outcomes. The **union** $A \cup B$ of two events A and B is the event that occurs when either A or B or both occur. The **intersection** $A \cap B$, also written AB, is the event that occurs when both events A and B occur. The events A and B are **mutually exclusive** if the occurrence of one of them excludes the occurrence of the other; that is, there is no outcome that is included in both A and B. The event $\overline{A}$, called the **complement** of A, is the event that occurs when A does

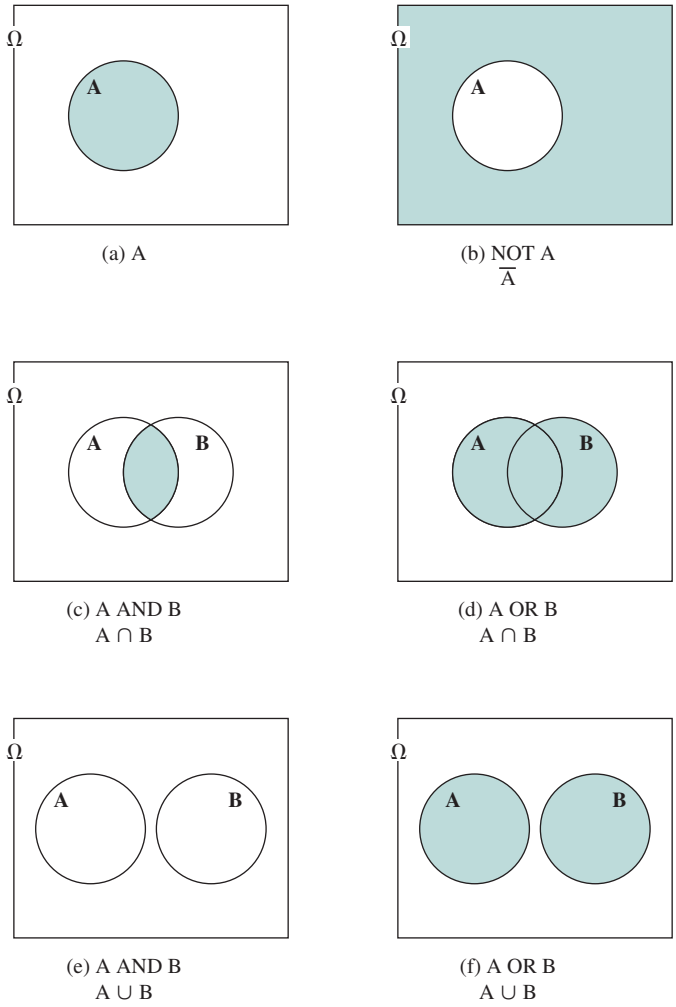


Figure 20.1 Venn Diagrams

not occur—that is, all outcomes in the sample space not included in A. These concepts are easily visualized with Venn diagrams, such as those shown in Figure 20.1. In each diagram, the shaded part corresponds to the expression below the diagram. Parts (c) and (d) correspond to cases in which A and B are not mutually exclusive; that is, some outcomes are defined as part of both events A and B. Parts (e) and (f) correspond to cases in which A and B are mutually exclusive. Note in these cases, the intersection of the two events is the empty set.

The common set of axioms used to define probability is as follows:

1. $0 \leq \Pr[A] \leq 1$ for each event A
2. $\Pr[\Omega] = 1$
3. $\Pr[A \cup B] = \Pr[A] + \Pr[B]$ if A and B are mutually exclusive

Axiom 3 can be extended to many events. For example, $\Pr[A \cup B \cup C] = \Pr[A] + \Pr[B] + \Pr[C]$ if A, B, and C are mutually exclusive. Note the axioms do not say anything about how probabilities are to be assigned to individual outcomes or events.

Based on these axioms, many laws can be derived. Here are some of the most important:

$$\Pr[\overline{A}) = 1 - \Pr[A]$$
$$\Pr[A \cap B] = 0 \text{ if } A \text{ and } B \text{ are mutually exclusive}$$
$$\Pr[A \cup B] = \Pr[A] + \Pr[B] - \Pr[A \cap B]$$
$$\Pr[A \cup B \cup C] = \Pr[A] + \Pr[B] + \Pr[C] - \Pr[A \cap B] - \Pr[A \cap C] - \Pr[B \cap C] + \Pr[A \cap B \cap C]$$

As an example, consider the throwing of a single die. This has six possible outcomes. The certain event is the event that occurs when any of the six die faces is on top. The union of the events {even} and {less than three} is the event {1 or 2 or 4 or 6}; the intersection of these events is the event {2}. The events {even} and {odd} are mutually exclusive. If we assume each of the six outcomes is equally likely and assign a probability of 1/6 to each outcome, it is easy to see that the three axioms are satisfied. We can apply the laws of probability as follows:

$$\Pr\{\text{even}\} = \Pr\{2\} + \Pr\{4\} + \Pr\{6\} = 1/2$$
$$\Pr\{\text{less than three}\} = \Pr\{1\} + \Pr\{2\} = 1/3$$
$$\Pr[\{\text{even}\} \cup \{\text{less than three}\}] = \Pr\{\text{even}\} + \Pr\{\text{less than three}\} - \Pr\{2\} = 1/2 + 1/3 - 1/6 = 2/3$$

Relative Frequency Definition The relative frequency approach uses the following definition of probability. Perform an experiment a number of times; each time is called a **trial**. For each trial, observe whether the event A occurs. Then the probability $\Pr[A]$ of an event A is the limit:

$$\Pr[A] = \lim_{n \to \infty} \frac{n_A}{n}$$

where n is the number of trials, and n_A is the number of occurrences of A.

For example, we could toss a coin many times. If the ratio of heads to total tosses hovers around 0.5 after a very large number of tosses, then we can assume that this is a fair coin, with equal probability of heads and tails.

Classical Definition For the classical definition, let N be the number of possible outcomes, with the restriction that all outcomes are equally likely, and N_A the number of outcomes in which event A occurs. Then the probability of A is defined as:

$$\Pr[A] = \frac{N_A}{N}$$

For example, if we throw one die, then N is 6 and there are three outcomes that correspond to the event {even}; hence $\Pr\{\text{even}\} = 3/6 = 0.5$. Here's a more complicated example: We roll two dice and want to determine the probability p that the sum is 7. You could consider the number of different sums that could be produced (2, 3, . . . , 12), which is 11, and conclude incorrectly that the probability is 1/11. We need to consider equally likely outcomes. For this purpose, we need to consider each combination of die faces, and we must distinguish between the first and second die. For example, the outcome (3, 4) must be counted separately from the outcome (4, 3). With this approach, there are 36 equally likely outcomes, and the favorable outcomes are the six pairs (1, 6), (2, 5), (3, 4), (4, 3), (5, 2), and (6, 1). Thus, $p = 6/36 = 1/6$.

Conditional Probability and Independence

We often want to know a probability that is conditional on some event. The effect of the condition is to remove some of the outcomes from the sample space. For example, what is the probability of getting a sum of 8 on the roll of two dice, if we know that the face of at least one die is an even number? We can reason as follows. Because one die is even and the sum is even, the second die must show an even number. Thus, there are three equally likely successful outcomes: (2, 6), (4, 4), and (6, 2), out of a total set of possibilities of $[36 - (\text{number of events with both faces odd})] = 36 - 3 \times 3 = 27$. The resulting probability is $3/27 = 1/9$.

Formally, the **conditional probability** of an event A assuming the event B has occurred, denoted by $\Pr[A\,|\,B]$, is defined as the ratio:

$$\Pr[A\,|\,B] = \frac{\Pr[AB]}{\Pr[B]}$$

where we assume $\Pr[B]$ is not zero.

In our example, $A = \{\text{sum of } 8\}$ and $B = \{\text{at least one die even}\}$. The quantity $\Pr[AB]$ encompasses all of those outcomes in which the sum is 8 and at least one die is even. As we have seen, there are three such outcomes. Thus, $\Pr[AB] = 3/36 = 1/12$. A moment's thought should convince you that $\Pr[B] = 3/4$. We can now calculate:

$$\Pr[A\,|\,B] = \frac{1/12}{3/4} = \frac{1}{9}$$

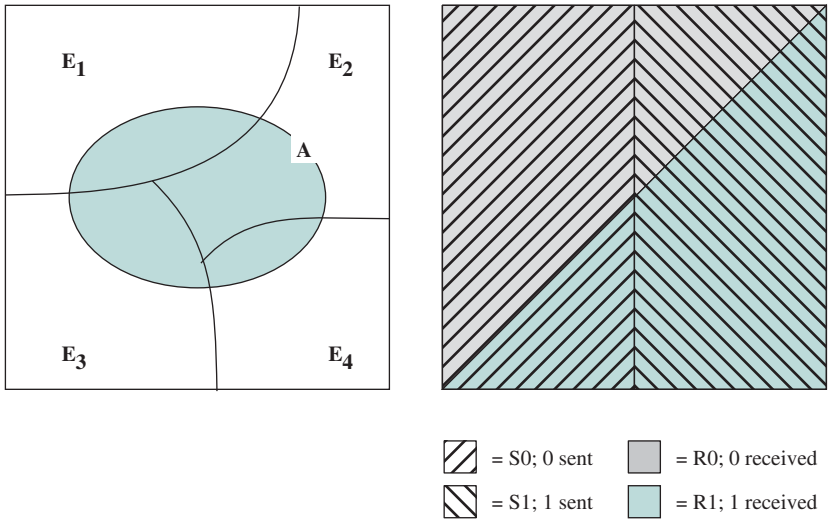
This agrees with our previous reasoning.

Two events A and B are called **independent** if $\Pr[AB] = \Pr[A]\Pr[B]$. It can easily be seen that if A and B are independent, $\Pr[A\,|\,B] = \Pr[A]$ and $\Pr[B\,|\,A] = \Pr[B]$.

Bayes's Theorem

We close this section with one of the most important results from probability theory, known as Bayes's Theorem. First, we need to state the total probability formula. Given a set of mutually exclusive events $E_1, E_2, \ldots, E_n$, such that the union of these events covers all possible outcomes, and given an arbitrary event A, then it can be shown that

$$\Pr[A] = \sum_{i=1}^{n} \Pr[A\,|\,E_i]\Pr[E_i] \qquad \textbf{(20.1)}$$



(a) Diagram to illustrate concepts (b) Example

Figure 20.2 Illustration of Total Probability and Bayes' Theorem

Bayes's Theorem may be stated as follows:

$$\Pr[E_i|A] = \frac{\Pr[A|E_i]\mathrm{P}[E_i]}{\Pr[A]} = \frac{\Pr[A|E_i]\mathrm{P}[E_i]}{\sum_{j=1}^{n}\Pr[A|E_j]\Pr[E_j]}$$

Figure 20.2a illustrates the concepts of total probability and Bayes's Theorem.

Bayes's Theorem is used to calculate *posterior odds*, that is, the probability that something really is the case, given evidence in favor of it. For example, suppose we are transmitting a sequence of 0s and 1s over a noisy transmission line. Let S0 and S1 be the events that a 0 is sent at a given time and a 1 is sent, respectively, and R0 and R1 be the events that a 0 is received and a 1 is received. Suppose we know the probabilities of the source, namely $\Pr[\mathrm{S1}] = p$ and $\Pr[\mathrm{S0}] = 1 - p$. Now the line is observed to determine how frequently an error occurs when a 1 is sent and when a 0 is sent, and the following probabilities are calculated: $\Pr[\mathrm{R0}|\mathrm{S1}] = p_a$ and $\Pr[\mathrm{R1}|\mathrm{S0}] = p_b$. If a 0 is received, we can then calculate the conditional probability of an error, namely the conditional probability that a 1 was sent given that a 0 was received, using Bayes's Theorem:

$$\Pr[\mathrm{S1}|\mathrm{R0}] = \frac{\Pr[\mathrm{R0}|\mathrm{S1}]\Pr[\mathrm{S1}]}{\Pr[\mathrm{R0}|\mathrm{S1}]\Pr[\mathrm{S1}] + \Pr[\mathrm{R0}|\mathrm{S0}]\Pr[\mathrm{S0}]} = \frac{p_a p}{p_a p + (1 - p_b)(1 - p)}$$

Figure 20.2b illustrates the preceding equation. In the figure, the sample space is represented by a unit square. Half of the square corresponds to S0 and half to S1, so $\Pr[\mathrm{S0}] = \Pr[\mathrm{S1}] = 0.5$. Similarly, half of the square corresponds to R0 and half to R1, so $\Pr[\mathrm{R0}] = \Pr[\mathrm{R1}] = 0.5$. Within the area representing S0, 1/4 of that area corresponds to R1, so $\Pr[\mathrm{R1}|\mathrm{S0}] = 0.25$. Other conditional probabilities are similarly evident.

20.2 RANDOM VARIABLES

A **random variable** is a mapping from the set of all possible events in a sample space under consideration to the real numbers. That is, a random variable associates a real number with each event. This concept is sometimes expressed in terms of an experiment with many possible outcomes; a random variable assigns a value to each such outcome. Thus, the value of a random variable is a random quantity. We give the following formal definition. A random variable X is a function that assigns a number to every outcome in a sample space and satisfies the following conditions:

1. The set $\{X \leq x\}$ is an event for every x.
2. $\Pr[X = \infty] = \Pr[X = -\infty] = 0$.

A random variable is **continuous** if it takes on an uncountably infinite number of distinct values. A random variable is **discrete** if it takes on a finite or countably infinite number of values.

Distribution and Density Functions

A continuous random variable X can be described by either its **distribution function** $F(x)$ or **density function** $f(x)$:

Distribution function: $F(x) = \Pr[X \leq x] \quad F(-\infty) = 0; \quad F(\infty) = 1$

Density function: $f(x) = \dfrac{d}{dx}F(x) \qquad F(x) = \displaystyle\int_{-\infty}^{x} f(y)dy \qquad \int_{-\infty}^{\infty} f(y)dy = 1$

For a discrete random variable, its probability distribution is characterized by

$$P_X(k) = \Pr[X = k] \sum_{\text{all } k} P_X(k) = 1$$

We are often concerned with some characteristic of a random variable rather than the entire distribution, such as shown in Table 20.1:

Table 20.1 Random Variable Characteristics

Mean value (also known as expected value or first moment)	$\begin{cases} \mathrm{E}[X] = \mu_X = \displaystyle\int_{-\infty}^{\infty} xf(x)dx & \text{continuous case} \\ \mathrm{E}[X] = \mu_X = \displaystyle\sum_{\text{all } k} k \Pr[x = k] & \text{discrete case} \end{cases}$
Second moment	$\begin{cases} \mathrm{E}[X^2] = \displaystyle\int_{-\infty}^{\infty} x^2 f(x)dx & \text{continuous case} \\ \mathrm{E}[X^2] = \displaystyle\sum_{\text{all } k} k^2 \Pr[x = k] & \text{discrete case} \end{cases}$
Variance	$\mathrm{Var}[X] = \mathrm{E}[(X - \mu_X)^2] = \mathrm{E}[X^2] - \mu_X^2$
Standard deviation	$\sigma_X = \sqrt{\mathrm{Var}[x]}$

The variance and standard deviation are measures of the dispersion of values around the mean. A high variance means the variable takes on more values relatively farther from the mean than for a low variance. It is easy to show that for any constant a:

$$\mathrm{E}[aX] = a\mathrm{E}[X]; \qquad \mathrm{Var}[aX] = a^2\mathrm{Var}[X]$$

The mean is known as a first-order statistic; the second moment and variance are second-order statistics. Higher-order statistics can also be derived from the probability density function.

Important Distributions

Several distributions that play an important role in queueing analysis are described next.

EXPONENTIAL DISTRIBUTION The exponential distribution with parameter $\lambda > 0$ is given by (see Figures 20.3a and 20.3b) and has the following distribution and density functions:

$$F(x) = 1 - e^{-\lambda x} \qquad f(x) = \lambda e^{-\lambda x} \qquad x \geq 0$$

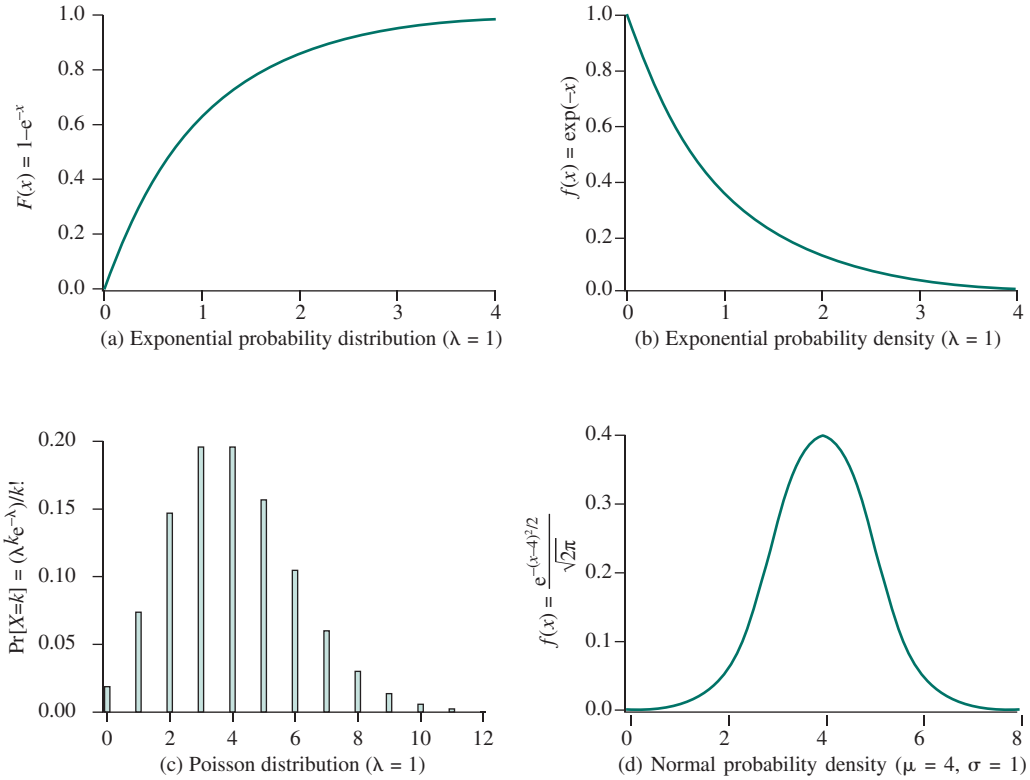


Figure 20.3 Some Probability Functions

The exponential distribution has the interesting property that its mean is equal to its standard deviation:

$$\mathrm{E}[X] = \sigma_X = \frac{1}{\lambda}$$

When used to refer to a time interval, such as a service time, this distribution is sometimes referred to as a random distribution. This is because, for a time interval that has already begun, each time at which the interval may finish is equally likely.

This distribution is important in queueing theory because we can often assume that the service time of a server in a queueing system is exponential. In the case of telephone traffic, the service time is the time for which a subscriber engages the equipment of interest. In a packet-switching network, the service time is the transmission time and is therefore proportional to the packet length. It is difficult to give a sound theoretical reason why service times should be exponential, but in many cases they are very nearly exponential. This is good news because it simplifies the queueing analysis immensely.

POISSON DISTRIBUTION Another important distribution is the Poisson distribution (see Figure 20.3c), with parameter $\lambda > 0$, which takes on values at the points $0, 1, \ldots$:

$$\Pr[X = k] = \frac{\lambda^k}{k!}\mathrm{e}^{-\lambda} \qquad k = 0, 1, 2 \ldots$$
$$\mathrm{E}[X] = \mathrm{Var}[X] = \lambda$$

If $\lambda < 1$, then $\Pr[X = k]$ is maximum for $k = 0$. If $\lambda > 1$ but not an integer, then $\Pr[X = k]$ is maximum for the largest integer smaller than λ; if λ is a positive integer, then there are two maxima at $k = \lambda$ and $k = \lambda - 1$.

The Poisson distribution is also important in queueing analysis because we must assume a Poisson arrival pattern to be able to develop the queueing equations (discussed in Chapter 21). Fortunately, the assumption of Poisson arrivals is usually valid.

The way in which the Poisson distribution can be applied to arrival rate is as follows. If items arrive at a queue according to a Poisson process, this may be expressed as:

$$\Pr[k \text{ items arrive in time interval } T] = \frac{(\lambda T)^k}{k!}\mathrm{e}^{-\lambda T}$$

$$\mathrm{E}[\text{number of items to arrive in time interval } T] = \lambda T$$

$$\text{Mean arrival rate, in items per second} = \lambda$$

Arrivals occurring according to a Poisson process are often referred to as random arrivals. This is because the probability of arrival of an item in a small interval is proportional to the length of the interval, and is independent of the amount of elapsed time since the arrival of the last item. That is, when items are arriving according to a Poisson process, an item is as likely to arrive at one instant as any other, regardless of the instants at which the other items arrive.

Another interesting property of the Poisson process is its relationship to the exponential distribution. If we look at the times between arrivals of items T_a (called

the interarrival times), then we find that this quantity obeys the exponential distribution:

$$\Pr[T_a < t] = 1 - e^{-\lambda t}$$
$$E[T_a] = \frac{1}{\lambda}$$

Thus, the mean interarrival time is the reciprocal of the arrival rate, as we would expect.

NORMAL DISTRIBUTION The normal distribution with parameters $\mu > 0$ and σ has the following density function (see Figure 20.3d) and distribution function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \qquad F(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-(y-\mu)^2/2\sigma^2} dy$$

with

$$E[X] = \mu$$
$$\text{Var}[X] = \sigma^2$$

An important result is the central limit theorem, which states that the distribution of the average of a large number of independent random variables will be approximately normal, almost regardless of their individual distributions. One key requirement is finite mean and variance. The central limit theorem plays a key role in statistics.

Multiple Random Variables

With two or more random variables, we are often concerned whether variations in one are reflected in the other. This subsection defines some important measures of dependence.

In general, the statistical characterization of multiple random variables requires a definition of their joint probability density function or joint probability distribution function:

Distribution: $F(x_1, x_2, \ldots x_n) = \Pr[X_1 \leq x_1, X_2 \leq x_2, \ldots, X_n \leq x_n]$

Density: $f(x_1, x_2, \ldots, x_n) = \frac{\partial^n}{\partial x_1 \partial x_2 \cdots \partial x_n} F(x_1, x_2, \ldots, x_n)$

Discrete distribution: $P(x_1, x_2, \ldots x_n) = \Pr[X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n]$

For any two random variables X and Y, we have

$$E[X + Y] = E[X] + E[Y]$$

Two continuous random variables X and Y are called (statistically) **independent** if $F(x, y) = F(x)F(y)$, and therefore $f(x, y) = f(x)f(y)$. If the random variables X and Y are discrete, then they are independent if $P(x, y) = P(x)P(y)$.

For independent random variables, the following relationships hold:

$$E[XY] = E[X] \times E[Y]$$
$$\text{Var}[X + Y] = \text{Var}[X] + \text{Var}[Y]$$

The **covariance** of two random variables X and Y is defined as follows:

$$\text{Cov}(X, Y) = \text{E}[(X - \mu_X)(Y - \mu_Y)] = \text{E}[XY] - \text{E}[X]\text{E}[Y]$$

If the variances of X and Y are finite, then their covariance is finite but may be positive, negative, or zero.

For finite variances of X and Y, the **correlation coefficient** of X and Y is defined as:

$$r(X, Y) = \frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y} \tag{20.2}$$

We can think of this as a measure of the linear dependence between X and Y, normalized to be relative to the amount of variability in X and Y. The following relationship holds:

$$-1 \leq r(X, Y) \leq 1$$

It is said X and Y are **positively correlated** if $r(X, Y) > 0$, that X and Y are **negatively correlated** if $r(X, Y) < 0$, and X and Y are **uncorrelated** if $r(X, Y) = \text{Cov}(X, Y) = 0$. If X and Y are independent random variables, then they are uncorrelated and $r(X, Y) = 0$. However, it is possible for X and Y to be uncorrelated but not independent (see Problem 20.12).

The correlation coefficient provides a measure of the extent to which two random variables are linearly related. If the joint distribution of X and Y is relatively concentrated around a straight line in the xy-plane that has a positive slope, then $r(X, Y)$ will typically be close to 1. This indicates that a movement in X will be matched by a movement of relatively similar magnitude and direction in Y. If the joint distribution of X and Y is relatively concentrated around a straight line that has a negative slope, then $r(X, Y)$ will typically be close to -1.

The following relationship is easily demonstrated:

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y) + 2\text{Cov}(X, Y)$$

If X and Y have the same variance σ^2, then the preceding can be rewritten as:

$$\text{Var}(X + Y) = 2\sigma^2(1 + r(X, Y))$$

If X and Y are uncorrelated [$r(X, Y) = 0$], then $\text{Var}(X + Y) = 2\sigma^2$. These results easily generalize to more than two variables: Consider a set of random variables $X_1, \ldots, X_N$, such that each has the same variance σ^2. Then

$$\text{Var}\left(\sum_{i=1}^{N} X_i\right) = \sigma^2\left(N + 2\sum_i \sum_{j<i} r(i, j)\right)$$

where $r(i, j)$ is shorthand for $r(X_i, X_j)$. Using the relationship $\text{Var}(X/N) = \text{Var}(X)/N^2$, we can develop an equation for the variance of the sample mean of a set of random variables:

$$\overline{X} = \frac{1}{N}\sum_{i=1}^{N} X_i$$

$$\text{Var}(\overline{X}) = \frac{\sigma^2}{N}\left(1 + \sum_i \sum_{j<i} r(i, j)\right)$$

If the X_i are mutually independent, then we have $\text{Var}(\overline{X}) = \dfrac{\sigma^2}{N}$.

20.3 ELEMENTARY CONCEPTS OF STOCHASTIC PROCESSES

A **stochastic process**, also called a **random process**, is a family of random variables $\{\mathbf{x}(t), t \in T\}$ indexed by a parameter t over some index set T. Typically, the index set is interpreted as the time dimension, and $\mathbf{x}(t)$ is a function of time. Another way to say this is a stochastic process is a random variable that is a function of time. A **continuous-time stochastic process** is one in which t varies continuously, typically over the nonnegative real line $\{\mathbf{x}(t), 0 \leq t < \infty\}$, although sometimes over the entire real line; whereas a **discrete-time stochastic process** is one in which t takes on discrete values, typically the positive integers $\{\mathbf{x}(t), t = 1, 2, \ldots\}$, although in some cases the range is the integers from $-\infty$ to $+\infty$.

Recall a random variable is defined as a function that maps the outcome of an experiment into a given value. With that in mind, the expression $\mathbf{x}(t)$ can be interpreted in several ways:

1. A family of time functions (t variable; all possible outcomes)
2. A single time function (t variable; one outcome)
3. A random variable (t fixed; all possible outcomes)
4. A single number (t fixed; one outcome)

The specific interpretation of $\mathbf{x}(t)$ is usually clear from the context.

A word about terminology. A **continuous-value stochastic process** is one in which the random variable $\mathbf{x}(t)$ with t fixed (case 3) takes on continuous values, whereas a **discrete-value stochastic process** is one in which the random variable at any time t takes on a finite or countably infinite number of values. A continuous-time stochastic process may be either continuous value or discrete value, and a discrete-time stochastic process may be either continuous value or discrete value.

As with any random variable, $\mathbf{x}(t)$ for a fixed value of t can be characterized by a probability distribution and a probability density. For continuous-value stochastic processes, these functions take the following form:

$$\text{Distribution function: } (x; t)F = \Pr[\mathbf{x}(t) \leq x] \quad F(-\infty; t) = 0; \quad F(\infty; t) = 1$$

$$\text{Density function: } f(x; t) = \frac{\partial}{\partial x}\mathrm{F}(x; t) \qquad F(x; t) = \int_{-\infty}^{x} f(y; t)dy \int_{-\infty}^{\infty} f(y; t)dy = 1$$

For discrete-value stochastic processes:

$$P_{\mathbf{x}(t)}(k) = \Pr[\mathbf{x}(t) = k] \qquad \sum_{\text{all } k} P_{\mathbf{x}(t)}(k) = 1$$

A full statistical characterization of a stochastic process must take into account the time variable. Using the first interpretation in the preceding list, a stochastic process $\mathbf{x}(t)$ comprises an infinite number of random variables, one for each t. To specify fully the statistics of the process, we would need to specify the joint probability density function of the variables $\mathbf{x}(t_1), \mathbf{x}(t_2), \ldots, \mathbf{x}(t_n)$ for all values of n $(1 \leq n < \infty)$ and all possible sampling times $(t_1, t_2, \ldots, t_n)$. For our purposes, we need not pursue this topic.

First- and Second-Order Statistics

The mean and variance of a stochastic process are defined in the usual way:

$$\mathrm{E}[\mathbf{x}(t)] = \mu(t) = \int_{-\infty}^{\infty} xf(x;t)dx \quad \text{continuous-value case}$$

$$\mathrm{E}[\mathbf{x}(t)] = \mu(t) = \sum_{\text{all } k} k \Pr[x(t) = k] \quad \text{discrete-value case}$$

$$\mathrm{E}[\mathbf{x}^2(t)] = \int_{-\infty}^{\infty} x^2 f(x;t)dx \quad \text{continuous-value case}$$

$$\mathrm{E}[\mathbf{x}^2(t)] = \sum_{\text{all } k} k^2 \Pr[x(t) = k] \quad \text{discrete-value case}$$

$$\mathrm{V}ar[\mathbf{x}(t)] = \sigma^2_{\mathbf{x}(t)} = \mathrm{E}[(\mathbf{x}(t) - \mu(t))^2] = \mathrm{E}[\mathbf{x}^2(t)] - \mu^2(t)$$

Note that, in general, the mean and variance of a stochastic process are functions of time. An important concept for our discussion is the **autocorrelation function** $R(t_1, t_2)$, which is the joint moment of the random variables $\mathbf{x}(t_1)$ and $\mathbf{x}(t_2)$:

$$R(t_1, t_2) = \mathrm{E}[\mathbf{x}(t_1)\mathbf{x}(t_2)]$$

As with the correlation function for two random variables introduced earlier, the autocorrelation is a measure of the relationship between the two time instances of a stochastic process. A related quantity is the **autocovariance**:

$$C(t_1, t_2) = \mathrm{E}[(\mathbf{x}(t_1) - \mu(t_1))(\mathbf{x}(t_2) - \mu(t_2))] = R(t_1, t_2) - \mu(t_1)\mu(t_2) \qquad \textbf{(20.3)}$$

Note the variance of $\mathbf{x}(t)$ is given by:

$$\mathrm{Var}[\mathbf{x}(t)] = C(t, t) = R(t, t) - \mu^2(t)$$

Finally, the **correlation coefficient** (see Equation 20.2) of $\mathbf{x}(t_1)$ and $\mathbf{x}(t_2)$ is called the normalized autocorrelation function of the stochastic process and can be expressed as:

$$\rho(t_1, t_2) = \frac{\mathrm{E}[(x(t_1) - \mu(t_1))(x(t_2) - \mu(t_2))]}{\sigma_1\sigma_2} = \frac{C(t_1, t_2)}{\sigma_1\sigma_2} \qquad \textbf{(20.4)}$$

Unfortunately, some texts and some of the literature refer to $\rho(t_1, t_2)$ as the autocorrelation function, so the reader must beware.

Stationary Stochastic Processes

In general terms, a **stationary stochastic process** is one in which the probability characteristics of the process do not vary as a function of time. There are several different precise definitions of this concept, but the one of most interest here is the concept of

wide sense stationary. A process is stationary in the wide sense (or weakly stationary) if its expected value is a constant and its autocorrelation function depends only on the time difference:

$$\begin{aligned} \mathrm{E}[\mathbf{x}(t)] &= \mu \\ R(t, t+\tau) &= R(t+\tau, t) = R(\tau) = R(-\tau) \quad \text{for all } t \end{aligned}$$

From these equalities, the following can be derived:

$$\begin{aligned} \mathrm{Var}[\mathbf{x}(t)] &= R(t,t) - \mu^2(t) = R(0) - \mu^2 \\ C(t, t+\tau) &= R(t, t+\tau) - \mu(t)\,\mu(t+\tau) = R(\tau) - \mu^2 = C(\tau) \end{aligned}$$

An important characteristic of $R(\tau)$ is that it measures the degree of dependence of one time instant of a stochastic process on other time instants. If $R(\tau)$ goes to zero exponentially fast as τ becomes large, then there is little dependence of one instant of a stochastic process on instants far removed in time. Such a process is called a **short memory process**, whereas if $R(\tau)$ remains substantial for large values of τ (decays to zero at a slower than exponential rate), the stochastic process is a **long memory process**.

Spectral Density

The **power spectrum**, or **spectral density**, of a stationary random process is the Fourier transform of its autocorrelation function:

$$S(w) = \int_{-\infty}^{\infty} R(\tau) e^{-\mathrm{j}w\tau} d\tau$$

where w is the frequency in radians ($w = 2\pi f$) and $\mathrm{j} = \sqrt{-1}$.

For a deterministic time function, the spectral density gives the distribution frequency of the power of the signal. For a stochastic process, $S(w)$ is the average density of power in the frequency components of $\mathbf{x}(t)$ in the neighborhood of w. Recall that one interpretation of $\mathbf{x}(t)$ is that of a single time function (t variable; one outcome). For that interpretation, the time function, as with any time function, is made up of a summation of frequency components, and its spectral density gives the relative power contributed by each component. If we view $\mathbf{x}(t)$ as a family of time functions (t variable; all possible outcomes), then the spectral density gives the average power in each frequency component, averaged over all possible time functions $\mathbf{x}(t)$.

The Fourier inversion formula gives the time function in terms of its Fourier transform:

$$R(\tau) = \frac{1}{2\pi} \int_{-\infty}^{\infty} S(w) e^{\mathrm{j}w\tau} dw$$

With $\tau = 0$, the preceding yields:

$$\frac{1}{2\pi} \int_{-\infty}^{\infty} S(w)\, dw = R(0) = \mathrm{E}[|\mathbf{x}(t)|^2]$$

Thus, the total area under $S(w)/2\pi$ equals the average power of the process $\mathbf{x}(t)$. Also note:

$$S(0) = \int_{-\infty}^{\infty} R(\tau)d\tau$$

$S(0)$ represents the direct-current (dc) component of the power spectrum and corresponds to the integral of the autocorrelation function. This component will be finite only if $R(\tau)$ decays as $\tau \rightarrow \infty$ sufficiently rapidly for the integral of $R(\tau)$ to be finite.

We can also express the power spectrum for a stochastic process that is defined at discrete points in time (discrete-time stochastic process). In this case, we have:

$$S(w) = \sum_{k=-\infty}^{\infty} R(k)e^{-jkw} \quad S(0) = \sum_{k=-\infty}^{\infty} R(k)$$

Again, $S(0)$ represents the dc component of the power spectrum and corresponds to the infinite sum of the autocorrelation function. This component will be finite only if $R(\tau)$ decays as $\tau \rightarrow \infty$ sufficiently rapidly for the summation to be finite.

Table 20.2 shows some interesting correspondences between the autocorrelation function and the power spectral density.

Independent Increments

A continuous-time stochastic process $\{\mathbf{x}(t), 0 \leq t < \infty\}$ is said to have independent increments if $\mathbf{x}(0) = 0$ and, for all choices of indexes $t_0 < t_1 < \ldots < t_n$, the n random variables

$$\mathbf{x}(t_1) - \mathbf{x}(t_0), \mathbf{x}(t_2) - \mathbf{x}(t_1), \ldots, \mathbf{x}(t_n) - \mathbf{x}(t_{n-1})$$

are independent. Thus, the amount of "movement" in a stochastic process in one time interval is independent of the movement in any other nonoverlapping time interval. The process is said to have stationary independent increments if, in addition, $\mathbf{x}(t_2 + h) - \mathbf{x}(t_1 + h)$ has the same distribution as $\mathbf{x}(t_2) - \mathbf{x}(t_1)$ for all choices of $t_2 > t_1$ and every $h > 0$.

Two properties of processes with stationary independent increments are noteworthy. If $\mathbf{x}(t)$ has stationary independent increments and $\mathrm{E}[\mathbf{x}(t)] = \mu(t)$ is a continuous function of time, then $\mu(t) = \mathrm{a} + \mathrm{b}t$, where a and b are constants. Also, if $\mathrm{Var}[\mathbf{x}(t) - \mathbf{x}(0)]$ is a continuous function of time, then for all s, $\mathrm{Var}[\mathbf{x}(s + t) - \mathbf{x}(s)] = \sigma^2 t$, where σ^2 is a constant.

Table 20.2 Autocorrelation Functions and Spectral Densities

Stationary Random Process	Autocorrelation Function	Power Spectral Density
$X(t)$	$R_X(\tau)$	$S_X(w)$
$aX(t)$	$a^2R_X(\tau)$	$\mathrm{a}^2S_X(w)$
$X'(t)$	$-\mathrm{d}^2R_X(\tau)/\mathrm{d}\tau^2$	$w^2S_X(w)$
$X^{(n)}(t)$	$(-1)^n\mathrm{d}^{2n}R_X(\tau)/\mathrm{d}\tau^{2n}$	$w^{2n}S_X(w)$
$X(t)\exp(\mathrm{j}w_0t)$	$\exp(\mathrm{j}w_0\tau)R_X(\tau)$	$S_X(w - w_0)$

Two processes that play a central role in the theory of stochastic processes, the Brownian motion process and the Poisson process, have independent increments. A brief introduction to both follows.

BROWNIAN MOTION PROCESS Brownian motion is the random movement of microscopic particles suspended in a liquid or gas, caused by collisions with molecules of the surrounding medium. This physical phenomenon is the basis for the definition of the Brownian motion stochastic process, also known as the Wiener process and the Wiener-Levy process.

Let us consider the function $B(t)$ for a particle in Brownian motion as denoting the displacement from a starting point in one dimension after time t. Consider the net movement of the particle in a time interval (s, t), which is long compared to the time between impacts. The quantity $B(t) - B(s)$ can be viewed as the sum of a large number of small displacements. By the central limit theorem, we can assume this quantity has a normal probability distribution.

If we assume the medium is in equilibrium, it is reasonable to assume the net displacement depends only on the length of the time interval and not on the time at which the interval begins. That is, the probability distribution of $B(t) - B(s)$ should be the same as $B(t + h) - B(s + h)$ for any $h > 0$. Finally, if the motion of the particle is due entirely to frequent random collisions, then the net displacements in nonoverlapping time intervals should be independent, and therefore $B(t)$ has independent increments.

With the foregoing reasoning in mind, we define a Brownian motion process $B(t)$ as one that satisfies the following conditions:

1. $\{ B(t), 0 \leq t < \infty \}$ has stationary independent increments.
2. For every $t > 0$, the random variable $B(t)$ has a normal distribution.
3. For all $t > 0$, $\mathrm{E}[B(t)] = 0$.
4. $B(0) = 0$.

The probability density of a Brownian motion process has the form:

$$f_B(x, t) = \frac{1}{\sigma\sqrt{2\pi t}} \mathrm{e}^{-x^2/2\sigma^2 t}$$

From this we have:

$$\mathrm{Var}[B(t)] = t; \qquad \mathrm{Var}[B(t) - B(s)] = |t - s|$$

Another important quantity is the autocorrelation of $B(t)$, expressed as $R_B(t_1, t_2)$. We derive this quantity in the following way. First, observe that for $t_4 > t_3 > t_2 > t_1$:

$$\begin{aligned} \mathrm{E}[(B(t_4) - B(t_3))(B(t_2) - B(t_1))] &= \mathrm{E}[B(t_4) - B(t_3)] \times \mathrm{E}[B(t_2) - B(t_1)] \\ &= (\mathrm{E}[B(t_4)] - \mathrm{E}[B(t_3)]) \times (\mathrm{E}[B(t_2)] - \mathrm{E}[B(t_1)]) \\ &= (0 - 0) \times (0 - 0) = 0 \end{aligned}$$

The first line of the preceding equation is true because the two intervals are nonoverlapping, and therefore the quantities $(B(t_4) - B(t_3))$ and $(B(t_2) - B(t_1))$ are independent, due to the assumption of independent increments. Recall that for

independent random variables X and Y, $\mathrm{E}[XY] = \mathrm{E}[X]\mathrm{E}[Y]$. Now consider the two intervals $(0, t_1)$ and (t_1, t_2), for $0 < t_1 < t_2$. These are nonoverlapping intervals, so

$$\begin{aligned} 0 &= \mathrm{E}[(B(t_2) - B(t_1))(B(t_1) - B(0))] \\ &= \mathrm{E}[(B(t_2) - B(t_1))B(t_1)] \\ &= \mathrm{E}[B(t_2)B(t_1)] - \mathrm{E}[B^2(t_1)] \\ &= \mathrm{E}[B(t_2)B(t_1)] - \mathrm{Var}[B(t_1)] \\ &= \mathrm{E}[B(t_2)B(t_1)] - t_1 \end{aligned}$$

Therefore,

$$R_B(t_1, t_2) = \mathrm{E}[B(t_1)B(t_2)] = t_1 \text{ where } t_1 < t_2$$

In general, then, the autocorrelation of $B(t)$ can be expressed as $R_B(t, s) = \min[t, s]$. Because $B(t)$ has zero mean, the autocovariance is the same as the autocorrelation. Thus, $C_B(t, s) = \min[t, s]$.

For any $t \geq 0$ and $\delta > 0$, the increment of a Brownian motion process, $B(t + \delta) - B(t)$, is normally distributed with mean 0 and variance δ. Thus,

$$\Pr[(B(t + \delta) - B(t)) \leq x] = \frac{1}{\sqrt{2\pi\delta}} \int_{-\infty}^{x} \mathrm{e}^{-y^2/2\delta} dy \qquad \textbf{(20.5)}$$

Note this distribution is independent of t and depends only on δ, consistent with the fact that $B(t)$ has stationary increments.

One useful way to visualize the Brownian motion process is as the limit of a discrete-time process. Let us consider a particle performing a random walk on the real line. At small time intervals τ, the particle randomly jumps a small distance δ to the left or right. We denote the position of the particle at time $k\tau$ as $X_\tau(k\tau)$. If positive and negative jumps are equally likely, then $X_\tau((k + 1)\tau)$ equals $X_\tau(k\tau) + \delta$ or $X_\tau(k\tau) - \delta$ with equal probability. If we assume $X_\tau(0) = 0$, then the position of the particle at time t is

$$X_\tau(t) = \delta(Y_1 + Y_2 \ldots + Y_{\lfloor t/\tau \rfloor})$$

where $Y_1, Y_2, \ldots$ are independent random variables with equal probability of being 1 or -1 and $\lfloor t/\tau \rfloor$ denotes the largest integer less than or equal to t/τ. It is convenient to normalize the step length δ as $\sqrt{\tau}$ so

$$X_\tau(t) = \sqrt{\tau}(Y_1 + Y_2 \ldots + Y_{\lfloor t/\tau \rfloor})$$

By the central limit theorem, for fixed t, if τ is sufficiently small then the sum in the preceding equation consists of many random variables, and therefore the distribution of $X_\tau(t)$ is approximately normal with mean 0 and variance t, because the Y_i have mean 0 and variance 1. Also, for fixed t and h, if τ is sufficiently small, then $X_\tau(t + h) - X_\tau(t)$ is approximately normal with mean 0 and variance h. Finally, we note the increments of $X_\tau(t)$ are independent. Thus, $X_\tau(t)$ is a discrete-time function that approximates Brownian motion. If we divide the time axis more finely, we improve the approximation. In the limit, this becomes a continuous-time Brownian motion process.

POISSON AND RELATED PROCESSES Recall that for random arrivals in time, we have the Poisson distribution:

$$\Pr[k \text{ items arrive in time interval } T] = \frac{(\lambda T)^k}{k!} e^{-\lambda T}$$

We can define a **Poisson counting process** $\{N(t), t \geq 0\}$ as follows:

1. $N(t)$ has stationary independent increments.
2. $N(0) = 0$.
3. For $0 < t_1 < t_2$, the quantity $N(t_2) - N(t_1)$ equals the number of points in the interval (t_1, t_2) and is Poisson distributed with mean $\lambda(t_1 - t_2)$.

Then we have the following probability functions for $N(t)$:

$$\Pr[N(t) = k] = \frac{(\lambda t)^k}{k!} e^{-\lambda t}$$

$$\mathrm{E}[N(t)] = \mathrm{Var}[N(t)] = \lambda t$$

Clearly, $N(t)$ is not stationary, because its mean is a function of time. Every time function of this stochastic process (one outcome) has the form of an increasing staircase with steps equal to 1, occurring at the random points t_i. Figure 20.4a gives an example of $N(t)$ for a specific outcome.

A stationary process related to the Poisson counting process is the **Poisson increment process**. For a Poisson counting process $N(t)$ with mean λt, and for a constant L $(L > 0)$, we can define the Poisson increment process $\mathrm{X}(t)$ as follows:

$$X(t) = \frac{N(t + L) - N(t)}{L}$$

$X(t)$ equals k/L, where k is the number of points in the interval $(t, t + L)$. The increment process derived from the counting process in Figure 20.4a is shown in Figure 20.4b. The following relationship holds.

$$\mathrm{E}[X(t)] = \frac{1}{L}\mathrm{E}[N(t + L)] - \frac{1}{L}\mathrm{E}[N(t)] = \lambda$$

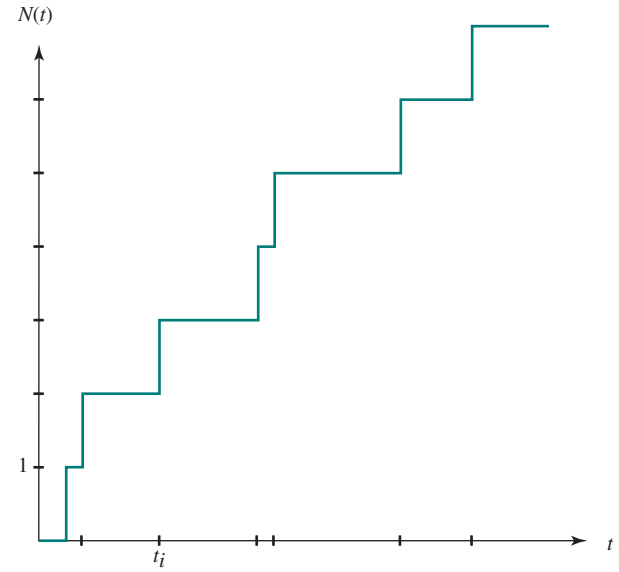
With a constant mean, $X(t)$ is a wide-sense stationary process and therefore has an autocorrelation function of a single variable, $R(\tau)$. It can be shown that this function is:

$$R(\tau) = \begin{cases} \lambda^2 & |\tau| > L \\ \lambda^2 + \dfrac{\lambda^2}{L}\left(1 - \dfrac{|\tau|}{L}\right) & |\tau| < L \end{cases} \tag{20.6}$$

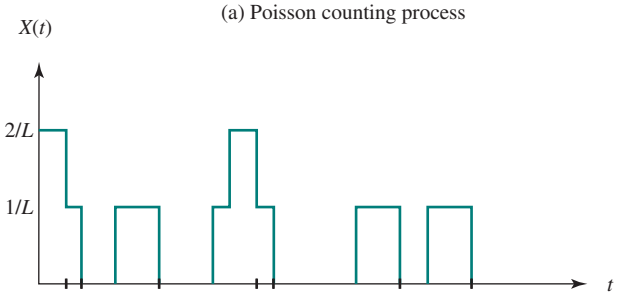
Thus, the correlation is greatest if the two time instants are within the interval length of each other, and it is a small constant value for greater time differences.

Ergodicity

For a stochastic process $\mathbf{x}(t)$, there are two types of "averaging" functions that can be performed: ensemble averages and time averages.



(a) Poisson counting process



(b) Poisson increment process

Figure 20.4 Poisson Processes

First, consider **ensemble averages**. For a constant value of t, $\mathbf{x}(t)$ is a single random variable with a mean, variance, and other distributional properties. For a given constant value C of t, the following measures exist:

$$\mathrm{E}[\mathbf{x}(\mathrm{C})] = \mu_{\mathbf{x}}(\mathrm{C}) = \int_{-\infty}^{\infty} xf(x; \mathrm{C})dx \qquad \text{continuous-value case}$$

$$\mathrm{E}[\mathbf{x}(\mathrm{C})] = \mu_{\mathbf{x}}(\mathrm{C}) = \sum_{\text{all } k} k \Pr[\mathbf{x}(\mathrm{C}) = k] \quad \text{discrete-value case}$$

$$\mathrm{Var}[\mathbf{x}(\mathrm{C})] = \sigma^2_{\mathbf{x}(\mathrm{C})} = \mathrm{E}[(\mathbf{x}(\mathrm{C}) - \mu_{\mathbf{x}}(\mathrm{C}))^2] = \mathrm{E}[\mathbf{x}(\mathrm{C})^2] - \mu^2_{\mathbf{x}}(\mathrm{C})$$

Each of these quantities is calculated over all values of $\mathbf{x}(t)$ for all possible outcomes. For a given random variable, the set of all possible outcomes is called an *ensemble*, and hence these are referred to as ensemble averages.

For time averages, consider a single outcome of $\mathbf{x}(t)$. This is a single deterministic function of t. Looking at $\mathbf{x}(t)$ in this way, we can consider what is the average value of the function over time. This **time average** is generally expressed as follows:

$$M_T = \frac{1}{2T}\int_{-T}^{T} \mathbf{x}(t)dt \quad \text{continuous-time case}$$

$$M_T = \frac{1}{T}\sum_{t=1}^{T} \mathbf{x}(t) \quad \text{discrete-time case}$$

Note M_T is a random variable, because the calculation of M_T for a single time function is a calculation for a single outcome.

A stationary process is said to be **ergodic** if time averages equal ensemble averages. Because $\mathrm{E}[\mathbf{x}(t)]$ is a constant for a stationary process, we have

$$\mathrm{E}[M_T] = \mathrm{E}[\mathbf{x}(t)] = \mu$$

Thus, we can say that a stationary process is ergodic if

$$\lim_{T\to\infty} \mathrm{Var}(M_T) = 0$$

In words, as the time average is taken over larger and larger time intervals, the value of the time average approaches the ensemble average.

The conditions under which a stochastic process is ergodic are beyond the scope of this book, but the assumption is generally made. Indeed, the assumption of ergodicity is essential to almost any mathematical model used for stationary stochastic processes. The practical importance of ergodicity is that in most cases, one does not have access to the ensemble of outcomes of a stochastic process or even to more than one outcome. Thus, the only means of obtaining estimates of the probabilistic parameters of the stochastic process is to analyze a single time function over a long period of time.

20.4 PROBLEMS

20.1 You are asked to play a game in which I hide a prize in one of three boxes (with equal probability for all three boxes) while you are out of the room. When you return, you have to guess which box hides the prize. There are two stages to the game. First, you indicate one of the three boxes as your choice. As soon as you do that, I open the lid of one of the other two boxes and I will always open an empty box. I can do this because I know where the prize is hidden. At this point, the prize must be in the box that you have chosen or in the other unopened box. You are now free to stick with your original choice or to switch to the other unopened box. You win the prize if your final selection is the box containing the prize. What is your best strategy? Should you (a) stay with your original choice, (b) switch to the other box, or (c) do either because it does not matter?

20.2 A patient has a test for some disease that comes back positive (indicating he has the disease). You are told that

- the accuracy of the test is 87% (i.e., if a patient has the disease, 87% of the time, the test yields the correct result, and if the patient does not have the disease, 87% of the time, the test yields the correct result)
- the incidence of the disease in the population is 1%

Given that the test is positive, how probable is it that the patient really has the disease?

20.3 A taxicab was involved in a fatal hit-and-run accident at night. Two cab companies, the Green and the Blue, operate in the city. You are told that:

- 85% of the cabs in the city are Green and 15% are Blue
- A witness identified the cab as Blue

The court tested the reliability of the witness under the same circumstances that existed on the night of the accident and concluded that the witness was correct in identifying the color of the cab 80% of the time. What is the probability that the cab involved in the incident was Blue rather than Green?

20.4 The birthday paradox is a famous problem in probability that can be stated as follows: What is the minimum value of K such that the probability is greater than 0.5 that at least two people in a group of K people have the same birthday? Ignore February 29 and assume each birthday is equally likely. We will do the problem in two parts:

a. Define $Q(K)$ as the probability that there are no duplicate birthdays in a group of K people. Derive a formula for $Q(K)$. *Hint:* First determine the number of different ways, N, that we can have K values with no duplicates.

b. Define $P(K)$ as the probability that there is at least one duplicate birthday in a group of K people. Derive this formula. What is the minimum value of K such that $P(K) > 0.5$? It may help to plot $P(K)$.

20.5 A pair of fair dice (the probability of each outcome is 1/6) is thrown. Let X be the maximum of the two numbers that comes up.

a. Find the distribution of X.

b. Find the expectation $\mathrm{E}[X]$, the variance $\mathrm{Var}[X]$, and the standard deviation σ_X.

20.6 A player tosses a fair die. If a prime number greater than 1 appears, he wins that number of dollars, but if a nonprime number appears, he loses that number of dollars.

a. Denote the player's gain or loss on one toss by the random variable X. Enumerate the distribution of X.

b. Is the game fair (i.e., $\mathrm{E}[X] = 0$)?

20.7 In the carnival game known as *chuck-a-luck*, a player pays an amount E as an entrance fee, selects a number between one and six, then rolls three dice. If all three dice show the number selected, the player is paid four times the entrance fee; if two dice show the number, the player is paid three times the entrance fee; and if only one die shows the number, the player is paid twice the entrance fee. If the selected number does not show up, the player is paid nothing. Let X denote the player's gain in a single play of this game, and assume the dice are fair.

a. Determine the probability function of X.

b. Compute $\mathrm{E}[X]$.

20.8 The mean and variance of X are 50 and 4, respectively. Evaluate the following:

a. The mean of X^2

b. The variance and standard deviation of $2X + 3$

c. The variance and standard deviation of $-X$

20.9 The continuous random variable R has a uniform density between 900 and 1,100, and 0 elsewhere. Find the probability that R is between 950 and 1,050.

20.10 Show that, all other things being equal, the greater the correlation coefficient of two random variables is, the greater the variance of their sum and the less the variance of their difference will be.

20.11 Suppose X and Y each have only two possible values, 0 and 1. Prove if X and Y are uncorrelated, then they are also independent.

20.12 Consider a random variable X with the following distribution: $\Pr[X = -1] = 0.25$; $\Pr[X = 0] = 0.5$; $\Pr[X = 1] = 0.25$. Let $Y = X^2$.

a. Are X and Y independent random variables? Justify your answer.

b. Calculate the covariance $\mathrm{Cov}(X, Y)$.

c. Are X and Y uncorrelated? Justify your answer.

20.13 An artificial example of a stochastic process is a deterministic signal $\mathbf{x}(t) = g(t)$. Determine the mean, variance, and autocorrelation of $\mathbf{x}(t)$.

20.14 Suppose $\mathbf{x}(t)$ is a stochastic process with

$$\mu(t) = 3 \quad R(t_1, t_2) = 9 + 4e^{-0.2|t_1 - t_2|}$$

Determine the mean, variance, and covariance of the following random variables: $Z = \mathbf{x}(5)$ and $W = \mathbf{x}(8)$.

20.15 Let $\{\mathbf{Z}_n\}$ be a set of uncorrelated real-valued random variables, each with a mean of 0 and a variance of 1. Define the moving average

$$\mathbf{Y}_n = \sum_{i=0}^{K} \alpha_i Z_{n-i}$$

for constants $\alpha_0, \alpha_1, \ldots, \alpha_K$. Show that $\mathbf{Y}$ is stationary and find its autocovariance function.

20.16 Let $\mathbf{X}_n = \mathbf{A}\cos(n\lambda) + \mathbf{B}\sin(n\lambda)$ where $\mathbf{A}$ and $\mathbf{B}$ are uncorrelated random variables, each with a mean of 0 and a variance of 1. Show that $\mathbf{X}$ is stationary with a spectrum containing exactly one point.

CHAPTER 21

QUEUEING ANALYSIS

Learning Objectives

After studying this chapter, you should be able to:

- Understand the characteristic behavior of queueing systems.
- Explain the value of queueing analysis.
- Explain the key features of single-server and multiserver queues.
- Analyze single-server queueing models.
- Analyze multiserver queueing models.
- Describe the effect of priorities on queueing performance.
- Understand the key concept relating to queueing networks.
- Understand the issues involved in estimating queueing model parameters.

Queueing[1] analysis is one of the most important tools for those involved with computer and network analysis. It can be used to provide approximate answers to a host of questions, such as:

- What happens to file retrieval time when disk I/O utilization goes up?
- Does response time change if both processor speed and the number of users on the system are doubled?
- How will performance be affected if the process scheduling algorithm includes priorities?
- Which disk scheduling algorithm produces the best average performance?

The number of questions that can be addressed with a queueing analysis is endless and touches on virtually every area in computer science. The ability to make such an analysis is an essential tool for those involved in this field.

Although the theory of queueing is mathematically complex, the application of queueing theory to the analysis of performance is, in many cases, remarkably straightforward. A knowledge of elementary statistical concepts (means and standard deviations) and a basic understanding of the applicability of queueing theory is all that is required. Armed with these, the analyst can often make a queueing analysis on the back of an envelope using readily available queueing tables, or with the use of simple computer programs that occupy only a few lines of code.

The purpose of this chapter is to provide a practical guide to queueing analysis. A subset, although a very important subset, of the subject is addressed. In the final section, pointers to additional references are provided. An annex to this paper reviews some elementary concepts in probability and statistics.

This chapter provides a practical guide to queueing analysis. A subset, although a very important subset, of the subject is addressed.

[1]Two spellings are in use: queueing and queueing. The vast majority of queueing theory researchers use *queueing*. The premier journal in this field is *Queueing systems: Theory and Applications*. On the other hand, most American dictionaries and spell checkers prefer the spelling *queueing*.

21.1 HOW QUEUES BEHAVE—A SIMPLE EXAMPLE

Before getting into the details of queueing analysis, let us look at a crude example that will give some feel for the topic. Consider a Web server that is capable of handling an individual request in an average of 1 millisecond. In fact, to make things simple, assume that the server handles each request in exactly 1 millisecond. Now, if the rate of arriving requests is 1 per millisecond (1,000 per second), then it seems sensible to state that the server can keep up with the load.

Suppose the requests arrive at a uniform rate of exactly one request each millisecond. When a request comes in, the server immediately handles the request. Just as the server completes the current request, a new request arrives and the server goes to work again.

Now let's take a more realistic approach and suppose the average arrival rate for requests is 1 per millisecond but that there is some variability. During any given 1 millisecond period, there may be no requests, or one, or multiple requests, but the average is still 1 per millisecond. Again, common sense would seem to indicate that the server could keep up. During busy times, when lots of requests bunch up, the server can store outstanding requests in a buffer. Another way of putting this is to say that arriving requests enter a queue to await service. During quiet times, the server can catch up and clear the buffer. In this case, the interesting design issue would seem to be: How big should the buffer be?

Tables 21.1 through 21.3 give a very rough idea of the behavior of this system. In Table 21.1, we assume an average arrival rate of 500 requests per second, which is half the capacity of the server. The entries in the table show the number of requests

Table 21.1 Queue Behavior with Normalized Arrival Rate of 0.5

Time	Input	Output	Queue
0	0	0	0
1	88	88	0
2	796	796	0
3	1627	1000	627
4	51	678	0
5	34	34	0
6	966	966	0
7	714	714	0
8	1276	1000	276
9	494	769	0
10	933	933	0
11	107	107	0
12	241	241	0
13	16	16	0
14	671	671	0

Table 21.1 Queue Behavior with Normalized Arrival Rate of 0.5 (*Continued*)

Time	Input	Output	Queue
15	643	643	0
16	812	812	0
17	262	262	0
18	218	218	0
19	1378	1000	378
20	507	885	0
21	15	15	0
22	820	820	0
23	1253	1000	253
24	307	559	0
25	540	540	0
26	190	190	0
27	500	500	0
28	96	96	0
29	943	943	0
30	105	105	0
31	183	183	0
32	447	447	0
33	542	542	0
34	166	166	0
35	165	165	0
36	490	490	0
37	510	510	0
38	877	877	0
39	37	37	0
40	163	163	0
41	104	104	0
42	42	42	0
43	291	291	0
44	645	645	0
45	363	363	0
46	134	134	0
47	920	920	0
48	1507	1000	507
49	598	1000	105
50	172	277	0
Average	499	499	43

that arrive each second, the number of requests served during that second, and the number of outstanding requests waiting in the buffer at the end of the second. After 50 seconds, the table shows an average buffer contents of 43 requests, with a peak of over 600 requests. In Table 21.2, the average arrival rate is increased to 95% of the server's capacity, that is, 950 requests per second, and the average buffer contents rises to 1859. This seems a little surprising: the arrival rate has gone up by less than a

Table 21.2 Queue Behavior with Normalized Arrival Rate of 0.95

Time	Input	Output	Queue
0	0	0	0
1	167	167	0
2	1512	1000	512
3	3091	1000	2603
4	97	1000	1700
5	65	1000	765
6	1835	1000	1600
7	1357	1000	1957
8	2424	1000	3381
9	939	1000	3320
10	1773	1000	4093
11	203	1000	3296
12	458	1000	2754
13	30	1000	1784
14	1275	1000	2059
15	1222	1000	2281
16	1543	1000	2824
17	498	1000	2322
18	414	1000	1736
19	2618	1000	3354
20	963	1000	3317
21	29	1000	2346
22	1558	1000	2904
23	2381	1000	4285
24	583	1000	3868
25	1026	1000	3894
26	361	1000	3255
27	950	1000	3205
28	182	1000	2387
29	1792	1000	3179
30	200	1000	2379

Table 21.2 Queue Behavior with Normalized Arrival Rate of 0.95 (*Continued*)

Time	Input	Output	Queue
31	348	1000	1727
32	849	1000	1576
33	1030	1000	1606
34	315	1000	921
35	314	1000	235
36	931	1000	166
37	969	1000	135
38	1666	1000	801
39	70	871	0
40	310	310	0
41	198	198	0
42	80	80	0
43	553	553	0
44	1226	1000	226
45	690	916	0
46	255	255	0
47	1748	1000	748
48	2863	1000	2611
49	1136	1000	2747
50	327	1000	2074
Average	948	907	1859

factor of 2, but the average buffer contents has gone up by more than a factor of 40. In Table 21.3, the average arrival rate is increased slightly, to 99% of capacity, which yields an average buffer contents of 2583. Thus, a tiny increase in average arrival rate results in an increase of almost 40% in the average buffer contents.

This crude example suggests that the behavior of a system with a queue may not accord with our intuition.

Table 21.3 Queue Behavior with Normalized Arrival Rate of 0.99

Time	Input	Output	Queue
0	0	0	0
1	174	174	0
2	1576	1000	576
3	3221	1000	2797
4	101	1000	1898
5	67	1000	965
6	1913	1000	1878

Time	Input	Output	Queue
7	1414	1000	2292
8	2526	1000	3818
9	978	1000	3796
10	1847	1000	4643
11	212	1000	3855
12	477	1000	3332
13	32	1000	2364
14	1329	1000	2693
15	1273	1000	2966
16	1608	1000	3574
17	519	1000	3093
18	432	1000	2525
19	2728	1000	4253
20	1004	1000	4257
21	30	1000	3287
22	1624	1000	3911
23	2481	1000	5392
24	608	1000	5000
25	1069	1000	5069
26	376	1000	4445
27	990	1000	4435
28	190	1000	3625
29	1867	1000	4492
30	208	1000	3700
31	362	1000	3062
32	885	1000	2947
33	1073	1000	3020
34	329	1000	2349
35	327	1000	1676
36	970	1000	1646
37	1010	1000	1656
38	1736	1000	2392
39	73	1000	1465
40	323	1000	788
41	206	994	0
42	83	83	0
43	576	576	0
44	1277	1000	277

Table 21.3 Queue Behavior with Normalized Arrival Rate of 0.99 (*Continued*)

Time	Input	Output	Queue
45	719	996	0
46	265	265	0
47	1822	1000	822
48	2984	1000	2806
49	1184	1000	2990
50	341	1000	2331
Average	988	942	2583

21.2 WHY QUEUEING ANALYSIS?

There are many cases when it is important to be able to project the effect of some change in a design: Either the load on a system is expected to increase, or a design change is contemplated. For example, an organization supports a number of terminals, personal computers, and workstations on a 100-Mbps local area network (LAN). An additional department in the building is to be cut over onto the network. Can the existing LAN handle the increased workload, or would it be better to provide a second LAN with a bridge between the two? There are other cases in which no facility exists but, on the basis of expected demand, a system design needs to be created. For example, a department intends to equip all of its personnel with a personal computer and to configure these into a LAN with a file server. Based on experience elsewhere in the company, the load generated by each PC can be estimated.

The concern is system performance. In an interactive or real-time application, often the parameter of concern is response time. In other cases, throughput is the principal issue. In any case, projections of performance are to be made on the basis of existing load information, or on the basis of estimated load for a new environment. A number of approaches are possible:

1. Do an after-the-fact analysis based on actual values.
2. Make a simple projection by scaling up from existing experience to the expected future environment.
3. Develop an analytic model based on queueing theory.
4. Program and run a simulation model.

Option 1 is no option at all: We will wait and see what happens. This leads to unhappy users and to unwise purchases. Option 2 sounds more promising. The analyst may take the position that it is impossible to project future demand with any degree of certainty. Therefore, it is pointless to attempt some exact modeling procedure. Rather, a rough-and-ready projection will provide ballpark estimates. The problem with this approach is that the behavior of most systems under a changing load is not what one would intuitively expect, as Section 21.1 suggests. If there is an environment in which there is a shared facility (e.g., a network, a transmission line, a time-sharing system), then the performance of that system typically responds in an exponential way to increases in demand.

Figure 21.1 is a representative example. The upper line shows what typically happens to user response time on a shared facility as the load on that facility increases. The load is expressed as a fraction of capacity. Thus, if we are dealing with an input from a disk that is capable of transferring 1,000 blocks per second, then a load of 0.5 represents a transfer of 500 blocks per second, and the response time is the amount of time it takes to retransmit any incoming block. The lower line is a simple projection based on a knowledge of the behavior of the system up to a load of 0.5. Note while things appear rosy when the simple projection is made, performance on the system will in fact collapse beyond a load of about 0.8–0.9.

Thus, a more exact prediction tool is needed. Option 3 is to make use of an analytic model, which is one that can be expressed as a set of equations that can be solved to yield the desired parameters (response time, throughput, etc.). For computer, operating system, and networking problems, and indeed for many practical real-world problems, analytic models based on queueing theory provide a reasonably good fit to reality. The disadvantage of queueing theory is that a number of simplifying assumptions must be made to derive equations for the parameters of interest.

The final approach is a simulation model. Here, given a sufficiently powerful and flexible simulation programming language, the analyst can model reality in great detail and avoid making many of the assumptions required of queueing theory. However, in most cases, a simulation model is not needed or at least is not advisable as a first step in the analysis. For one thing, both existing measurements and projections of future load carry with them a certain margin of error. Thus, no matter how good

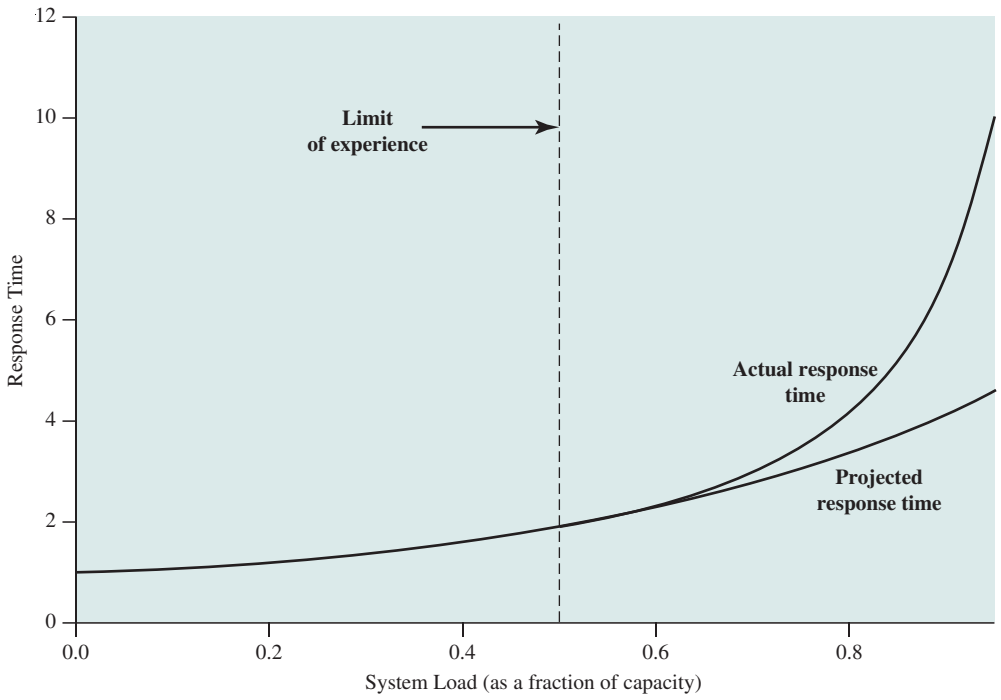


Figure 21.1 Projected Versus Actual Response Time

the simulation model, the value of the results is limited by the quality of the input. For another, despite the many assumptions required of queueing theory, the results that are produced often come quite close to those that would be produced by a more careful simulation analysis. Furthermore, a queueing analysis can literally be accomplished in a matter of minutes for a well-defined problem, whereas simulation exercises can take days, weeks, or longer to program and run.

Accordingly, it behooves the analyst to master the basics of queueing theory.

21.3 QUEUEING MODELS

The Single-Server Queue

The simplest queueing system is depicted in Figure 21.2. The central element of the system is a server, which provides some service to items. Items from some population of items arrive at the system to be served. If the server is idle, an item is served immediately. Otherwise, an arriving item joins a waiting line.[2] When the server has completed serving an item, the item departs. If there are items waiting in the queue, one is immediately dispatched to the server. The server in this model can represent anything that performs some function or service for a collection of items. For example, a processor provides service to processes, a transmission line provides a transmission service to packets or frames of data, and an I/O device provides a read or write service for I/O requests.

QUEUE PARAMETERS Figure 21.2 also illustrates some important parameters associated with a queueing model. Items arrive at the facility at some average rate λ (items arriving per second). Some examples of items arriving include packets arriving at a router and calls arriving at a telephone exchange. At any given time, a certain number of items will be waiting in the waiting line (zero or more); the average number waiting is w, and the mean time that an item must wait is T_w. T_w is averaged

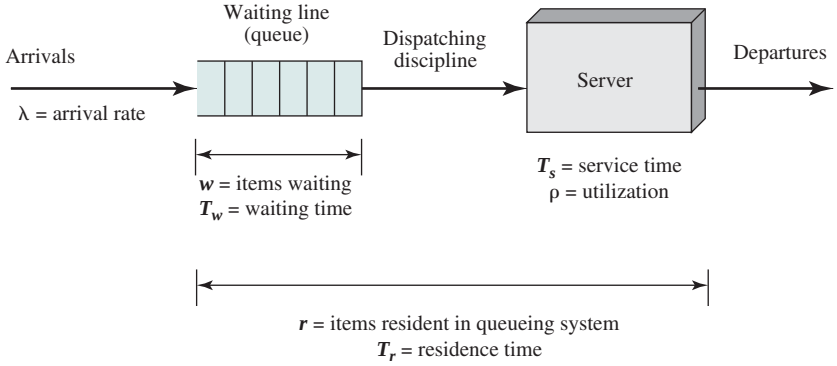


Figure 21.2 Queueing System Structure and Parameters for Single-Server Queue

[2]The waiting line is referred to as a queue in some treatments in the literature; it is also common to refer to the entire system as a queue. Unless otherwise noted, we use the term *queue* to mean waiting line.

over all incoming items, including those that do not wait at all. The server handles incoming items with an average service time T_s; this is the time interval between the dispatching of an item to the server and the departure of that item from the server. Utilization, ρ, is the fraction of time that the server is busy, measured over some interval of time. Finally, two parameters apply to the system as a whole. The average number of items resident in the system, including the item being served (if any) and the items waiting (if any), is r; and the average time that an item spends in the system, waiting and being served, is T_r; we refer to this as the *mean residence time*.[3]

If we assume the capacity of the queue is infinite, then no items are ever lost from the system; they are just delayed until they can be served. Under these circumstances, the departure rate equals the arrival rate. As the arrival rate, which is the rate of traffic passing through the system, increases, the utilization increases and with it, congestion. The queue becomes longer, increasing waiting time. At $\rho = 1$, the server becomes saturated, working 100% of the time. So long as utilization is less than 100%, the server can keep up with arrivals, so the average departure rate equals the average arrival rate. Once the server is saturated, working 100% of the time, the departure rate remains constant, no matter how great the arrival rate becomes. Thus, the theoretical maximum input rate that can be handled by the system is:

$$\lambda_{\max} = \frac{1}{T_s}$$

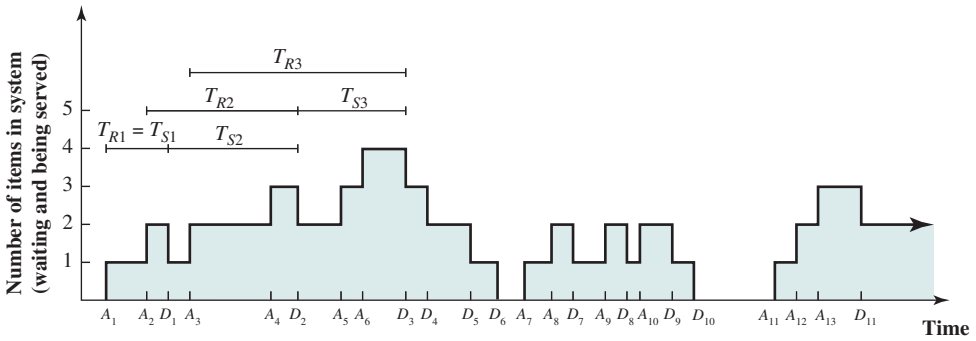
However, queues become very large near system saturation, growing without bound when $\rho = 1$. Practical considerations, such as response time requirements or buffer sizes, usually limit the input rate for a single server to 70–90% of the theoretical maximum.

ILLUSTRATION OF KEY FEATURES It is helpful to have an illustration of the processes involved in queueing. Figure 21.3 shows an example realization of a queueing process, with the total number of items in the system plotted against time. The shaded areas represent time periods in which the server is busy. On the time axis are marked two types of events: the arrival of item j at time A_j and the completion of service of item j at time D_j, when the item departs the system. The time that item j spends in the system is $T_{Rj} = D_j - A_j$; the actual service time for item j is denoted by T_{Sj}.

In this example, T_{R1} is composed entirely of the service time T_{S1} for the first item, because when item 1 arrives the system is empty and it can go straight into service. T_{R2} is composed of the time that item 2 waits for service $(D_1 - A_2)$ plus its service time T_{S2}. Similarly, $T_{R3} = (D_3 - A_3) = (D_3 - D_2) + (D_2 - A_3) = T_{S3} + (D_2 - A_3)$. However, item n may depart before the arrival of item $n + 1$, (e.g., $D_6 < A_7$), so the general expression is $T_{Rn+1} = T_{Sn+1} + \text{MAX}[0, D_n - A_{n+1}]$.

MODEL CHARACTERISTICS Before deriving any analytic equations for the queueing model, certain key characteristics of the model must be chosen. The following are the typical choices, usually reasonable in a data communications context:

[3]Again, in some of the literature, this is referred to as the mean queueing time, while other treatments use mean queueing time to mean the average time spent waiting in the waiting line (before being served).



For item i:

A_i = Arrival time
D_i = Departure time
T_{Ri} = Residence time
T_{Si} = Service time

Figure 21.3 Example of a Queueing Process

- **Item population:** We assume items arrive from a source population so large that it can be viewed as infinite. The effect of this assumption is that the arrival rate is not altered as items enter the system. If the population is finite, then the population available for arrival is reduced by the number of items currently in the system; this would typically reduce the arrival rate proportionally. Networking and server problems can usually be handled with an infinite-population assumption.
- **Queue size:** We assume an infinite queue size. Thus, the queue can grow without bound. With a finite queue, items can be lost from the system; that is, if the queue is full and additional items arrive, some items must be discarded. In practice, any queue is finite, but in many cases, this makes no substantive difference to the analysis. We will address this issue briefly later in this chapter.
- **Dispatching discipline:** When the server becomes free, and if there is more than one item waiting, a decision must be made as to which item to dispatch next. The simplest approach is first-in-first-out (FIFO), also known as first-come-first-served (FCFS); this discipline is what is normally implied when the term *queue* is used. Another possibility is last-in-last-out (LIFO). A common approach is a dispatching discipline based on relative priority. For example, a router may use QoS (quality of service) information to give preferential treatment to some packets. We will discuss dispatching based on priority subsequently. One dispatching discipline that you might encounter in practice is based on service time. For example, a process scheduler may choose to dispatch processes on the basis of shortest first (to allow the largest number of processes to be granted time in a short interval) or longest first (to minimize processing time relative to service time). Unfortunately, a discipline based on service time is very difficult to model analytically.

Table 21.4 summarizes the notation that is used in Figure 21.2 and introduces some other useful parameters. In particular, we are often interested in the variability of various parameters, and this is neatly captured in the standard deviation.

Table 21.4 Notation for Queueing Systems

λ	= arrival rate; mean number of arrivals per second
T_s	= mean service time for each arrival; amount of time being served, not counting time waiting in the queue
σ_{Ts}	= standard deviation of service time
ρ	= utilization; fraction of time facility (server or servers) is busy
u	= traffic intensity
r	= mean number of items in system, waiting and being served
R	= number of items in system, waiting and being served
T_r	= mean time an item spends in system (residence time)
T_R	= time an item spends in system (residence time)
σ_r	= standard deviation of r
σ_{Tr}	= standard deviation of T_r
w	= mean number of items waiting to be served
σ_w	= standard deviation of w
T_w	= mean waiting time (including items that have to wait and items with waiting time = 0)
T_d	= mean waiting time for items that have to wait
N	= number of servers
$m_x(y)$	= the yth percentile; that value of y below which x occurs y percent of the time

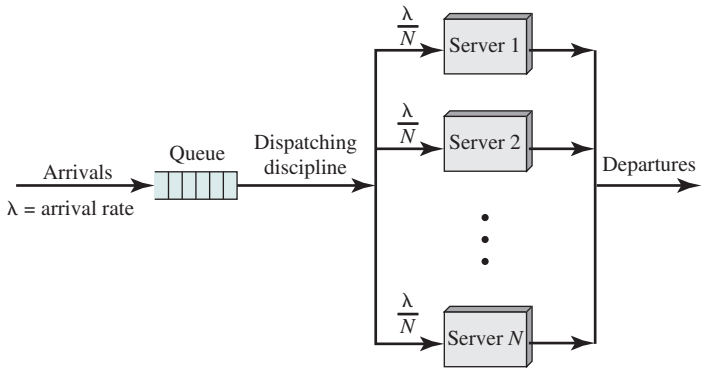
The Multiserver Queue

Figure 21.4a shows a generalization of the simple model we have been discussing for multiple servers, all sharing a common queue. If an item arrives and at least one server is available, then the item is immediately dispatched to that server. It is assumed all servers are identical; thus, if more than one server is available, the selection of a particular server for a waiting item has no effect on service time. If all servers are busy, a queue begins to form. As soon as one server becomes free, an item is dispatched from the queue using the dispatching discipline in force.

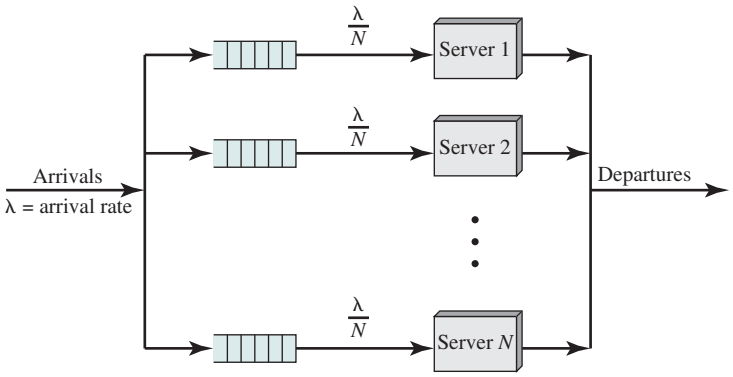
With the exception of utilization, all of the parameters illustrated in Figure 21.2 carry over to the multiserver case with the same interpretation. If we have N identical servers, then ρ is the utilization of each server, and we can consider $N\rho$ to be the utilization of the entire system; this latter term is often referred to as the *traffic intensity*, u. Thus, the theoretical maximum utilization is $N \times 100\%$, and the theoretical maximum input rate is:

$$\lambda_{\max} = \frac{N}{T_s}$$

The key characteristics typically chosen for the multiserver queue correspond to those for the single-server queue. That is, we assume an infinite population and an infinite queue size, with a single infinite queue shared among all servers. Unless otherwise stated in this discussion, the dispatching discipline is FIFO. For the multiserver case, if all servers are assumed identical, the selection of a particular server for a waiting item has no effect on service time.



(a) Multiserver queue



(b) Multiple Single-server queues

Figure 21.4 Multiserver Versus Multiple Single-Server Queues

By way of contrast, Figure 21.4b shows the structure of multiple single-server queues. As we shall see, this apparently minor change in structure has significant impact on performance.

Basic Queueing Relationships

To proceed much further, we have to make some simplifying assumptions. These assumptions risk making the models less valid for various real-world situations. Fortunately, in most cases, the results will be sufficiently accurate for planning and design purposes.

There are, however, some relationships that are true in the general case, and these are illustrated in Table 21.5. By themselves, these relationships are not particularly helpful, although they can be used to answer a few basic questions. For example, consider a spy from Burger King trying to figure out how many people are inside the McDonald's across the way. He can't sit inside the McDonald's all day, so he has to determine an answer just based on observing the traffic in and out of the building.

Over the course of the day, he observes that on average 32 customers per hour go into the restaurant. He notes certain people and finds that on average a customer stays inside 12 minutes. Using Little's formula, the spy deduces that there are on average 6.4 customers in McDonald's at any given time (6.4 = 32 customers per hour × 0.2 hours per customer).

It would be useful at this point to gain an intuitive grasp of the equations in Table 21.5. For the equation $\rho = \lambda T_s$, consider that for an arrival rate of λ, the average time between arrivals is $1/\lambda = T$. If T is greater than T_s, then during a time interval T, the server is only busy for a time T_s for a utilization of $T_s/T = \lambda T_s$. Similar reasoning applies in the multiserver case to yield $\rho = (\lambda T_s)/N$.

To understand Little's formula, consider the following argument, which focuses on the experience of a single item. When the item arrives, it will find on average w items waiting ahead of it. When the item leaves the queue behind it to be serviced, it will leave behind on average the same number of items in the queue, namely w. To see this, note while the item is waiting, the line in front of it shrinks until the item is at the front of the line; meanwhile, additional items arrive and get in line behind this item. When the item leaves the queue to be serviced, the number of items behind it, on average, is w, because w is defined as the average number of items waiting. Further, the average time that the item was waiting for service is T_w. Since items arrive at a rate of λ, we can reason that in the time T_w, a total of λT_w items must have arrived. Thus, $w = \lambda T_w$. Similar reasoning can be applied to the relationship $r = \lambda T_r$.

Turning to the last equation in the first column of Table 21.5, it is easy to observe that the time that an item spends in the system is the sum of the time waiting for service plus the time being served. Thus, on average, $T_r = T_w + T_s$. The last equations in the second and third columns are easily justified. At any time, the number of items in the system is the sum of the number of items waiting for service plus the number of items being served. For a single server, the average number of items being served is ρ. Therefore, $r = w + \rho$ for a single server. Similarly, $r = w + N\rho$ for N servers.

Assumptions

The fundamental task of a queueing analysis is as follows: Given the following information as input:

- Arrival rate
- Service time
- Number of servers

Table 21.5 Some Basic Queueing Relationships

General	Single Server	Multiserver
$r = \lambda T_r$ Little's formula $w = \lambda T_w$ Little's formula $T_r = T_w + T_s$	$\rho = \lambda T_s$ $r = w + \rho$	$\rho = \frac{\lambda T_s}{N}$ $u = \lambda T_s = \rho N$ $r = w + N\rho$

provide as output information concerning:

- Items waiting
- Waiting time
- Items in residence
- Residence time

What specifically would we like to know about these outputs? Certainly we would like to know their average values (w, T_w, r, T_r). In addition, it would be useful to know something about their variability. Thus, the standard deviation of each would be useful (σ_r, σ_{T_r}, σ_w, σ_{T_w}). Other measures may also be useful. For example, to design a buffer associated with a router or multiplexer, it might be useful to know for what buffer size the probability of overflow is less than 0.001. That is, what is the value of N such that Pr[items waiting $< N$] $= 0.999$?

To answer such questions in general requires complete knowledge of the probability distribution of the interarrival times (time between successive arrivals) and service time. Furthermore, even with that knowledge, the resulting formulas are exceedingly complex. Thus, to make the problem tractable, we need to make some simplifying assumptions.

The most important of these assumptions concerns the arrival rate. We assume the interarrival times are exponential, which is equivalent to saying that the number of arrivals in a period t obeys the Poisson distribution, which is equivalent to saying that the arrivals occur randomly and independent of one another. This assumption is almost invariably made. Without it, most queueing analysis is impractical, or at least quite difficult. With this assumption, it turns out that many useful results can be obtained if only the mean and standard deviation of the arrival rate and service time are known. Matters can be made even simpler and more detailed results can be obtained if it is assumed the service time is exponential or constant.

A convenient notation, called **Kendall's notation**, has been developed for summarizing the principal assumptions that are made in developing a queueing model. The notation is X/Y/N, where X refers to the distribution of the interarrival times, Y refers to the distribution of service times, and N refers to the number of servers. The most common distributions are denoted as follows:

G = general distribution of interarrival times or service times

GI = general distribution of interarrival times with the restriction that interarrival times are independent

M = negative exponential distribution

D = deterministic arrivals or fixed-length service

Thus, M/M/1 refers to a single-server queueing model with Poisson arrivals (exponential interarrival times) and exponential service times.

21.4 SINGLE-SERVER QUEUES

Table 21.6a provides some equations for single-server queues that follow the M/G/1 model. That is, the arrival rate is Poisson and the service time is general. Making use of a scaling factor, A, the equations for some of the key output variables are straightforward. Note the key factor in the scaling parameter is the ratio of the standard deviation of service time to the mean. No other information about the service time is needed. Two special cases are of some interest. When the standard deviation is equal to the mean, the service time distribution is exponential (M/M/1). This is the simplest case, and the easiest one for calculating results. Table 21.6b shows the simplified versions of equations for the standard deviation of r and T_r, plus some other parameters of interest. The other interesting case is a standard deviation of service time equal to zero, that is, a constant service time (M/D/1). The corresponding equations are shown in Table 21.6c.

Figures 21.5 and 21.6 plot values of average queue size and residence time versus utilization for three values of σ_{T_s}/T_s. This latter quantity is known as the **coefficient of variation** and gives a normalized measure of variability. Note the poorest performance is exhibited by the exponential service time, and the best by a constant service time. In many cases, one can consider the exponential service time to be a worst case, so an analysis based on this assumption will give conservative results. This is nice, because tables are available for the M/M/1 case and values can be looked up quickly.

What value of σ_{T_s}/T_s is one likely to encounter? We can consider four regions:

- **Zero:** This is the rare case of constant service time. For example, if all transmitted packets are of the same length, they would fit this category.
- **Ratio less than 1:** Because this ratio is better than the exponential case, using M/M/1 tables will give queue sizes and times that are slightly larger than they should be. Using the M/M/1 model would give answers on the safe side. An example of this category might be a data entry application for a particular form.
- **Ratio close to 1:** This is a common occurrence and corresponds to exponential service time. That is, service times are essentially random. Consider message lengths to a computer terminal: A full screen might be 1920 characters, with message sizes varying over the full range. Airline reservations, file lookups on inquiries, shared LAN, and packet-switching networks are examples of systems that often fit this category.
- **Ratio greater than 1:** If you observe this, you need to use the M/G/1 model and not rely on the M/M/1 model. A common occurrence of this is a bimodal distribution, with a wide spread between the peaks. An example is a system that experiences many short messages, many long messages, and few in between.

The same consideration applies to the arrival rate. For a Poisson arrival rate, the interarrival times are exponential, and the ratio of standard deviation to mean is 1. If the observed ratio is much less than one, then arrivals tend to be evenly spaced (not

Table 21.6 Formulas for Single-Server Queues

Assumptions:	1. Poisson arrival rate. 2. Dispatching discipline does not give preference to items based on service times. 3. Formulas for standard deviation assume first-in-first-out dispatching. 4. No items are discarded from the queue.

(a) General Service Times (M/G/1)

$$A = \frac{1}{2}\left[1 + \left(\frac{\sigma_{T_s}}{T_s}\right)^2\right]$$

$$r = \rho + \frac{\rho^2 A}{1-\rho}$$

$$w = \frac{\rho^2 A}{1-\rho}$$

$$T_r = T_s + \frac{\rho T_s A}{1-\rho}$$

$$T_w = \frac{\rho T_s A}{1-\rho}$$

(b) Exponential Service Times (M/M/1)

$$r = \frac{\rho}{1-\rho} \qquad w = \frac{\rho^2}{1-\rho}$$

$$T_r = \frac{T_s}{1-\rho} \qquad T_w = \frac{\rho T_s}{1-\rho}$$

$$\sigma_r = \frac{\sqrt{\rho}}{1-\rho} \qquad \sigma_{T_r} = \frac{T_s}{1-\rho}$$

$$\Pr[R = N] = (1-\rho)\rho^N$$

$$\Pr[R \le N] = \sum_{i=0}^{N} (1-\rho)\rho^i$$

$$\Pr[T_R \le T] = 1 - e^{-(1-\rho)t/T_s}$$

$$m_{T_r}(y) = T_r \times \ln\left(\frac{100}{100-y}\right)$$

$$m_{T_w}(y) = \frac{T_w}{\rho} \times \ln\left(\frac{100\rho}{100-y}\right)$$

(c) Constant Service Times (M/D/1)

$$r = \frac{\rho^2}{2(1-\rho)} + \rho$$

$$w = \frac{\rho^2}{2(1-\rho)}$$

$$T_r = \frac{T_s(2-\rho)}{2(1-\rho)}$$

$$T_w = \frac{\rho T_s}{2(1-\rho)}$$

$$\sigma_r = \frac{1}{1-\rho}\sqrt{\rho - \frac{3\rho^2}{2} + \frac{5\rho^3}{6} - \frac{\rho^4}{12}}$$

$$\sigma_{T_r} = \frac{T_s}{1-\rho}\sqrt{\frac{\rho}{3} - \frac{\rho^2}{12}}$$

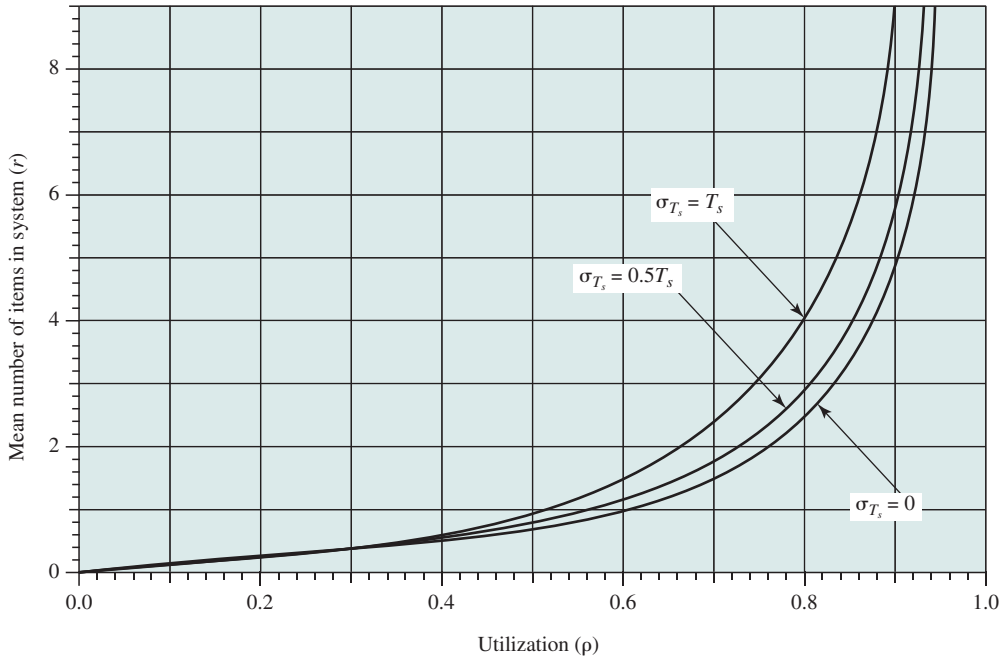


Figure 21.5 **Mean Number of Items in System for Single-Server Queue**

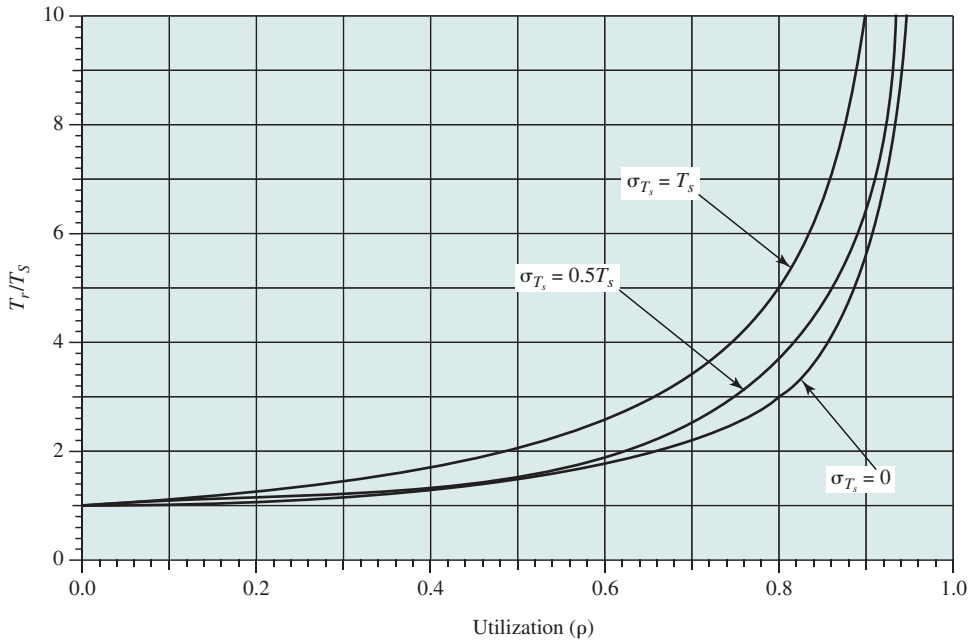


Figure 21.6 **Mean Residence Time for Single-Server Queue**

much variability), and the Poisson assumption will overestimate queue sizes and delays. On the other hand, if the ratio is greater than 1, then arrivals tend to cluster and congestion becomes more acute.

21.5 MULTISERVER QUEUES

Table 21.7 lists formulas for some key parameters for the multiserver case. Note the restrictiveness of the assumptions. Useful congestion statistics for this model have been obtained only for the case of M/M/N, where the exponential service times are identical for the N servers.

Note the presence of the Erlang C function in nearly all of the equations. This is the probability that all servers are busy at a given instant; equivalently, this is the probability that the number of items in the system (waiting and being served) is greater than or equal to the number of servers. The equation has the form

$$C(N, \rho) = \frac{1 - K(N, \rho)}{1 - \rho K(N, \rho)}$$

where K is known as the Poisson ratio function. Because C is a probability, its value is always between zero and one. As can be seen, this quantity is a function of the number of servers and the utilization. This expression turns up frequently in queueing calculations. Tables of values are readily found, or a computer program must be used. Note for a single-server system, this equation simplifies to $C(1, \rho) = \rho$.

21.6 EXAMPLES

Let us look at a few examples to get some feel for the use of these equations.

Database Server

Consider a LAN with 100 personal computers and a server that maintains a common database for a query application. The average time for the server to respond to a query is 0.6 seconds, and the standard deviation is estimated to equal the mean. At peak times, the query rate over the LAN reaches 20 queries per minute. We would like to answer the following questions:

- What is the average response time ignoring line overhead?
- If a 1.5-second response time is considered the maximum acceptable, what percent growth in message load can occur before the maximum is reached?
- If 20% more utilization is experienced, will response time increase by more or less than 20%?

Table 21.7 Formulas for Multiserver Queues (M/M/N)

Assumptions:	1. Poisson arrival rate. 2. Exponential service times. 3. All servers equally loaded. 4. All servers have same mean service time. 5. First-in-first-out dispatching. 6. No items are discarded from the queue.

$$K = \frac{\sum_{I=0}^{N-1} \frac{(N\rho)^I}{I!}}{\sum_{I=0}^{N} \frac{(N\rho)^I}{I!}} \qquad \text{Poisson ratio function}$$

$$\text{Erlang C function} = \text{Probability that all servers are busy} = C = \frac{1 - K}{1 - \rho K}$$

$$r = C\frac{\rho}{1-\rho} + N\rho \qquad w = C\frac{\rho}{1-\rho}$$

$$T_r = \left(\frac{C}{N}\right)\frac{T_s}{1-\rho} + T_s \qquad T_w = \left(\frac{C}{N}\right)\frac{T_s}{1-\rho}$$

$$\sigma_{T_r} = \frac{T_s}{N(1-\rho)}\sqrt{C(2-C) + N^2(1-\rho)^2}$$

$$\sigma_w = \frac{1}{1-\rho}\sqrt{C\rho(1+\rho-C\rho)}$$

$$\Pr[T_W > t] = Ce^{-N(1-\rho)t/T_s}$$

$$m_{T_w}(y) = \frac{T_s}{N(1-\rho)}\ln\left(\frac{100C}{100-y}\right)$$

$$T_d = \frac{T_s}{N(1-\rho)}$$

Assume an M/M/1 model, with the database server being the server in the model. We ignore the effect of the LAN, assuming that its contribution to the delay is negligible. Facility utilization is calculated as:

$$\begin{aligned}\rho &= \lambda T_s \\ &= (20 \text{ arrivals per minute})(0.6 \text{ seconds per transmission})/(60 \text{ s/min}) \\ &= 0.2\end{aligned}$$

The first value, average response time, is easily calculated:

$$\begin{aligned}T_r &= T_s/(1-\rho) \\ &= 0.6/(1-0.2) = 0.75 \text{ seconds}\end{aligned}$$

The second value is more difficult to obtain. Indeed, as worded, there is no answer because there is a nonzero probability that some instances of response time will exceed 1.5 seconds for any value of utilization. Instead, let us say we would like

90% of all responses to be less than 1.5 seconds. Then, we can use the equation from Table 21.6b:

$$m_{T_r}(y) = T_r \times \ln(100/(100 - y))$$
$$m_{T_r}(90) = T_r \times \ln(10) = \frac{T_s}{1 - \rho} \times 2.3 = 1.5 \text{ seconds}$$

We have $T_s = 0.6$. Solving for ρ yields $\rho = 0.08$. In fact, utilization would have to decline from 20% to 8% to put 1.5 seconds at the 90th percentile.

The third part of the question is to find the relationship between increases in load versus response time. Because a facility utilization of 0.2 is down in the flat part of the curve, response time will increase more slowly than utilization. In this case, if facility utilization increases from 20% to 40%, which is a 100% increase, the value of T_r goes from 0.75 seconds to 1.0 second, which is an increase of only 33%.

Calculating Percentiles

Consider a configuration in which packets are sent from computers on a LAN to systems on other networks. All of these packets must pass through a router that connects the LAN to a wide area network and hence to the outside world. Let us look at the traffic from the LAN through the router. Packets arrive with a mean arrival rate of 5 per second. The average packet length is 144 octets, and it is assumed packet length is exponentially distributed. Line speed from the router to the wide area network is 9,600 bps. The following questions are asked:

1. What is the mean residence time for the router?
2. How many packets are in the router, including those waiting for transmission and the one currently being transmitted (if any), on the average?
3. Same question as (2), for the 90th percentile.
4. Same question as (2), for the 95th percentile.

$\lambda = 5$ packets per second
$T_s = (144 \text{ octets} \times 8 \text{ bits per octet})/9{,}600 \text{ bps} = 0.12$ seconds
$\rho = \lambda T_s = 5 \times 0.12 = 0.6$
$T_r = T_s/(1 - \rho) = 0.3$ seconds Mean residence time
$r = \rho/(1 - \rho) = 1.5$ packets Mean number of resident items

To obtain the percentiles, we use the equation from Table 21.6b:

$$\Pr[R = N] = (1 - \rho)\rho^N$$

To calculate the yth percentile of queue size, we write the preceding equation in cumulative form:

$$\frac{y}{100} = \sum_{k=0}^{m_r(y)} (1 - \rho)\rho^k = 1 - \rho^{1+m_r(y)}$$

Here $m_r(y)$ represents the maximum number of packets in the queue expected y percent of the time. That is, $m_r(y)$ is that value below which R occurs y percent of the

time. In the form given, we can determine the percentile for any queue size. We wish to do the reverse: Given y, find $m_r(y)$. So, taking the logarithm of both sides:

$$m_r(y) = \frac{\ln\left(1 - \frac{y}{100}\right)}{\ln \rho} - 1$$

If $m_r(y)$ is fractional, take the next higher integer; if it is negative, set it to zero. For our example, $\rho = 0.6$ and we wish to find $m_r(90)$ and $m_r(95)$:

$$m_r(90) = \frac{\ln(1 - 0.90)}{\ln(0.6)} - 1 = 3.5$$

$$m_r(95) = \frac{\ln(1 - 0.95)}{\ln(0.6)} - 1 = 4.8$$

Thus, 90% of the time there are fewer than 4 packets in the queue, and 95% of the time there are fewer than 5 packets. If we were designing to a 95th percentile criterion, a buffer would have to be provided to store at least 5 packets.

Tightly-Coupled Multiprocessor

Let us consider the use of multiple tightly-coupled processors in a single computer system. One of the design decisions had to do with whether processes are dedicated to processors. If a process is permanently assigned to one processor from activation until its completion, then a separate short-term queue is kept for each processor. In this case, one processor can be idle, with an empty queue, while another processor has a backlog. To prevent this situation, a common queue can be used. All processes go into one queue and are scheduled to any available processor. Thus, over the life of a process, the process may be executed on different processors at different times.

Let us try to get a feel for the performance speed-up to be achieved by using a common queue. Consider a system with five processors, and the average amount of processor time provided to a process while in the Running state is 0.1 second. Assume the standard deviation of service time is observed to be 0.094 second. Because the standard deviation is close to the mean, we will assume exponential service time. Also assume processes are arriving at the Ready state at the rate of 40 per second.

SINGLE-SERVER APPROACH If processes are evenly distributed among the processors, then the load for each processor is 40/5 = 8 processes per second. Thus,

$$\begin{aligned}\rho &= \lambda T_s \\ &= 8 \times 0.1 = 0.8\end{aligned}$$

The residence time is then easily calculated:

$$t_r = \frac{T_s}{1 - \rho} = \frac{0.1}{0.2} = 0.5 \text{ sec}$$

MULTISERVER APPROACH Now assume a single Ready queue is maintained for all processors. We now have an aggregate arrival rate of 40 processes per second. However, the facility utilization is still 0.8 ($\lambda T_s/M$). To calculate the residence time from the formula in Table 21.7, we need to first calculate the Erlang C function. If you have not programmed the parameter, it can be looked up in a table under a facility utilization of 0.8 for five servers to yield $C = 0.554$. Substituting,

$$T_r = (0.1) + \frac{(0.544)(0.1)}{5(1 - 0.8)} = 0.1544$$

So the use of multiserver queue has reduced average residence time from 0.5 seconds down to 0.1544 seconds, which is greater than a factor of 3. If we look at just the waiting time, the multiserver case is 0.0544 seconds compared to 0.4 seconds, which is a factor of 7.

Although you may not be an expert in queueing theory, you now know enough to be annoyed when you have to wait in a line at a multiple single-server queue facility.

A Multiserver Problem

An engineering firm provides each of its analysts with a personal computer, all of which are hooked up over a LAN to a database server. In addition, there is an expensive, stand-alone graphics workstation that is used for special-purpose design tasks. During the course of a typical eight-hour day, 10 engineers will make use of the workstation and spend an average of 30 minutes at a session.

SINGLE-SERVER MODEL The engineers complain to their manager that the wait for using the workstation is long, often an hour or more, and are asking for more workstations. This surprises the manager since the utilization of the workstation is only 5/8 ($10 \times 1/2 = 5$ hours out of 8). To convince the manager, one of the engineers performs a queueing analysis. The engineer makes the usual assumptions of an infinite population, random arrivals, and exponential service times, none of which seem unreasonable for rough calculations. Using the equations in Tables 21.5 and 21.6b, the engineer gets:

$T_w = \frac{\rho T_s}{1 - \rho} = 50$ minutes	Average time an engineer spends waiting for the workstation
$m_{T_w}(90) = \frac{T_w}{\rho} \times \ln(10\rho) = 146.6$ minutes	90th percentile waiting time
$\lambda = \frac{10}{8 \times 60} = 0.021$ engineers/minute	Arrival rate of engineers
$w = \lambda T_w = 1.0416$ engineers	Average number of engineers waiting

These figures show that indeed the engineers do have to wait an average of almost an hour to use the workstation, and that in 10% of the cases, an engineer has to wait well over two hours. Even if there is a significant error in the estimate, say

20%, the waiting time is still far too long. Furthermore, if an engineer can do no useful work while waiting for the workstation, then a little over one engineer-day is being lost per day.

MULTISERVER MODEL The engineers have convinced the manager of the need for more workstations. They would like the mean waiting time not to exceed 10 minutes, with the 90th percentile value not to exceed 15 minutes. This concerns the manager, who reasons that if one workstation results in a waiting time of 50 minutes, then five workstations will be required to get the average down to 10 minutes.

The engineers set to work to determine how many workstations are required. There are two possibilities: Put additional workstations in the same room as the original one (multiserver queue) or scatter the workstations to various rooms on various floors (multiple single-server queues). First, we look at the multiserver case and consider the addition of a second workstation in the same room. Let's assume that the addition of the new workstation, which reduces waiting time, does not affect the arrival rate (10 engineers per day). Then the available service time is 16 hours in an eight-hour day with a demand of five hours (10 engineers $\times$ 0.5 hours), giving a utilization of $5/16 = 0.3125$. Using the equations in Table 21.7:

$C(2, \rho) = C(2, 0.3125) = 0.1488$	Probability that both servers are busy
$T_w = \dfrac{CT_s}{N(1-\rho)} = 3.247$ minutes	Average time an engineer spends waiting for a workstation
$m_{T_w}(90) = \dfrac{T_s}{2(1-\rho)} \ln(10C) = 8.67$ minutes	90th percentile waiting time
$w = \lambda T_w = 0.07$ engineers	Average number of engineers waiting

With this arrangement, the probability that an engineer who wishes to use a workstation must wait is less than 0.15 and the average wait is just a little over three minutes, with the 90th percentile wait of less than nine minutes. Despite the manager's doubts, the multiserver arrangement with two workstations easily meets the design requirement.

All of the engineers are housed on two floors of the building, so the manager wonders whether it might be more convenient to place one workstation on each floor. If we assume the traffic to the two workstations is about evenly split, then there are two M/M/1 queues, each with a λ of five engineers per eight-hour day. This yields:

$\rho = \lambda T_s = 0.3125$	Utilization of one server
$T_w = \dfrac{\rho T_s}{1-\rho} = 13.64$ minutes	Average time an engineer spends waiting for the workstation
$m_{T_w}(90) = \dfrac{T_w}{\rho} \times \ln(10\rho) = 49.73$ minutes	90th percentile waiting time
$w = \lambda T_w = 0.142$ engineers	Average number of engineers waiting

Table 21.8 Summary of Calculations for Multiserver Example

Workstations	System	ρ	T_w	$m_{Tw}(90)$
1	M/M/1	0.625	50	146.61
2	M/M/2	0.3125	3.25	8.67
3	M/M/1's	0.3125	13.64	49.73
4	M/M/1's	0.15625	5.56	15.87
5	M/M/1's	0.125	4.29	7.65

This performance is significantly worse than the multiserver model and does not meet the design criteria. Table 21.8 summarizes the results and also shows the results for four and five separate workstations. Note to meet the design goal, five separate workstations are needed compared to only two multiserver workstations.

21.7 QUEUES WITH PRIORITIES

So far, we have considered queues in which items are treated in a first-come-first-served basis. There are many cases in both networking and operating system design in which it is desirable to use priorities. Priorities may be assigned in a variety of ways. For example, priorities may be assigned on the basis of traffic type. If it turns out that the average service time for the various traffic types is identical, then the overall equations for the system are not changed, although the performance seen by the different traffic classes will differ.

An important case is one in which priority is assigned on the basis of average service time. Often, items with shorter expected service times are given priority over items with longer service times. For example, a router may assign a higher priority to a stream of voice packets than a stream of data packets, and typically, the voice packets would be much shorter than the data packets. With this kind of scheme, performance is improved for higher-priority traffic.

Table 21.9 shows the formulas that apply when we assume two priority classes with different service times for each class. These results are easily generalized to any number of priority classes.

To see the effects of the use of priority, let us consider a simple example of a data stream consisting of a mixture of long and short packets being transmitted by a packet-switching node and that the rate of arrival of the two types of packets is equal. Suppose both packets have lengths that are exponentially distributed, and the long packets have a mean packet length of 10 times the short packets. In particular, let us assume a 64-Kbps transmission link and the mean packet lengths are 80 and 800 octets. Then the two service times are 0.01 and 0.1 seconds. Also assume the arrival rate for each type is 8 packets per second. So the shorter packets are not held up by the longer packets, let us assign the shorter packets a higher priority. Then:

$$\rho_1 = 8 \times 0.01 = 0.08 \qquad \rho_2 = 8 \times 0.1 = 0.8 \qquad \rho = 0.88$$

Table 21.9 Formulas for Single-Server Queues with Two Priority Categories

Assumptions:	**1.** Poisson arrival rate. **2.** Priority 1 items are serviced before priority 2 items. **3.** First-in-first-out dispatching for items of equal priority. **4.** No item is interrupted while being served. **5.** No items leave the queue (lost calls delayed).

(a) General Formulas

$$\lambda = \lambda_1 + \lambda_2$$
$$\rho_1 = \lambda_1 T_{s1};\ \rho_2 = \lambda_2 T_{s2}$$
$$\rho = \rho_1 + \rho_2$$
$$T_s = \frac{\lambda_1}{\lambda} T_{s1} + \frac{\lambda_2}{\lambda} T_{s2}$$
$$T_r = \frac{\lambda_1}{\lambda} T_{r1} + \frac{\lambda_2}{\lambda} T_{r2}$$

(b) Exponential Service Times

$$w_1 = \frac{\rho_1(\rho_1 T_{s1} + \rho_2 T_{s2})}{T_{s1}(1 - \rho_1)}$$
$$w_2 = w_1 \frac{\lambda_2}{\lambda_1(1 - \rho)}$$
$$T_{r1} = T_{s1} + \frac{\rho_1 T_{s1} + \rho_2 T_{s2}}{1 - \rho_1}$$
$$T_{r2} = T_{s2} + \frac{T_{r1} - T_{s1}}{1 - \rho}$$

$$T_{r1} = 0.01 + \frac{0.08 \times 0.01 + 0.8 \times 0.1}{1 - 0.08} = 0.098 \text{ seconds}$$
$$T_{r2} = 0.1 + \frac{0.098 - 0.01}{1 - 0.88} = 0.833 \text{ seconds}$$
$$T_r = 0.5 \times 0.098 + 0.5 \times 0.833 = 0.4655 \text{ seconds}$$

So we see the higher-priority packets get considerably better service than the lower-priority packets.

21.8 NETWORKS OF QUEUES

In a distributed environment, isolated queues are unfortunately not the only problem presented to the analyst. Often, the problem to be analyzed consists of several interconnected queues. Figure 21.7 illustrates this situation, using nodes to represent queues and the interconnecting lines to represent traffic flow.

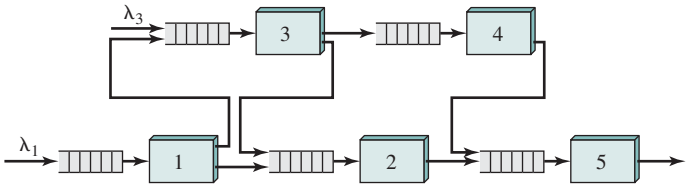


Figure 21.7 Example of a Network of Queues

Two elements of such a network complicate the methods shown so far:

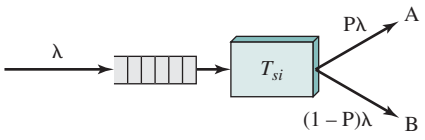
- The partitioning and merging of traffic, as illustrated by nodes 1 and 5, respectively, in the figure
- The existence of queues in tandem, or series, as illustrated by nodes 3 and 4

No exact method has been developed for analyzing general queueing problems that have the aforementioned elements. However, if the traffic flow is Poisson and the service times are exponential, an exact and simple solution exists. In this section, we first examine the two elements listed previously, then present the approach to queueing analysis.

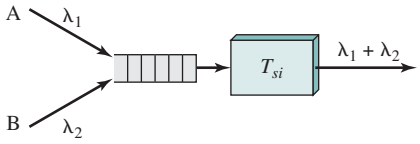
Partitioning and Merging of Traffic Streams

Suppose traffic arrives at a queue with a mean arrival rate of λ, and that there are two paths, A and B, by which an item may depart (see Figure 21.8a). When an item is serviced and departs the queue, it does so via path A with probability P and via path B with probability $(1 - P)$. In general, the traffic distribution of streams A and B will differ from the incoming distribution. However, if the incoming distribution is Poisson, then the two departing traffic flows also have Poisson distributions, with mean rates of $P\lambda$ and $(1 - P)\lambda$.

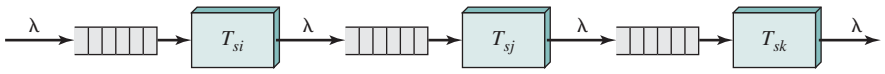
A similar situation exists for traffic merging (see Figure 21.8b). If two Poisson streams with mean rates of λ_1 and λ_2 are merged, the resulting stream is Poisson with a mean rate of $\lambda_1 + \lambda_2$.



(a) Traffic partitioning



(b) Traffic merging



(c) Simple tandem queue

Figure 21.8 Elements of Queueing Networks

Both of these results generalize to more than two departing streams for partitioning and more than two arriving streams for merging.

Queues in Tandem

Figure 21.8c is an example of a set of single-server queues in tandem: The input for each queue except the first is the output of the previous queue. Assume the input to the first queue is Poisson. Then, if the service time of each queue is exponential and the queues are of infinite capacity, the output of each queue is a Poisson stream statistically identical to the input. When this stream is fed into the next queue, the delays at the second queue are the same as if the original traffic had bypassed the first queue and fed directly into the second queue. Thus, the queues are independent and may be analyzed one at a time. Therefore, the mean total delay for the tandem system is equal to the sum of the mean delays at each stage.

This result can be extended to the case where some or all of the nodes in tandem are multiserver queues.

Jackson's Theorem

Jackson's theorem can be used to analyze a network of queues. The theorem is based on three assumptions:

1. The queueing network consists of m nodes, each of which provides an independent exponential service.
2. Items arriving from outside the system to any one of the nodes arrive with a Poisson rate.
3. Once served at a node, an item goes (immediately) to one of the other nodes with a fixed probability, or out of the system.

Jackson's theorem states that in such a network of queues, each node is an independent queueing system, with a Poisson input determined by the principles of partitioning, merging, and tandem queueing. Thus, each node may be analyzed separately from the others using the M/M/1 or M/M/N model, and the results may be combined by ordinary statistical methods. Mean delays at each node may be added to derive system delays, but nothing can be said about the higher moments of system delays (e.g., standard deviation).

Jackson's theorem appears attractive for application to packet-switching networks. One can model the packet-switching network as a network of queues. Each packet represents an individual item. We assume each packet is transmitted separately and, at each packet-switching node in the path from source to destination, the packet is queued for transmission on the next length. The service at a queue is the actual transmission of the packet and is proportional to the length of the packet.

The flaw in this approach is that a condition of the theorem is violated: Namely, it is not the case that the service distributions are independent. Because the length of a packet is the same at each transmission link, the arrival process to each queue is correlated to the service process. However, Kleinrock [KLEI76] has demonstrated that, because of the averaging effect of merging and partitioning, assuming independent service times provides a good approximation.

Application to a Packet-Switching Network[4]

Consider a packet-switching network, consisting of nodes interconnected by transmission links, with each node acting as the interface for zero or more attached systems, each of which functions as a source and destination of traffic. The external workload that is offered to the network can be characterized as:

$$\gamma = \sum_{j=1}^{N} \sum_{k=1}^{N} \gamma_{jk}$$

where

γ = total workload in packets per second

γ_{jk} = workload between source j and destination k

N = total number of sources and destinations

Because a packet may traverse more than one link between source and destination, the total internal workload will be higher than the offered load:

$$\lambda = \sum_{i=1}^{L} \lambda_i$$

where

λ = total load on all of the links in the network

λ_i = load on link i

L = total number of links

The internal load will depend on the actual path taken by packets through the network. We will assume a routing algorithm is given such that the load on the individual links, λ_i, can be determined from the offered load, γ_{jk}. For any particular routing assignment, we can determine the average number of links that a packet will traverse from these workload parameters. Some thought should convince you that the average length for all paths is given by:

$$\text{E[number of links in a path]} = \frac{\lambda}{\gamma}$$

Now, our objective is to determine the average delay, T, experienced by a packet through the network. For this purpose, it is useful to apply Little's formula (see Table 21.5). For each link in the network, the average number of items waiting and being served for that link is given by:

$$r_i = \lambda_i T_{ri}$$

where T_{ri} is the yet-to-be-determined queueing delay at each queue. Suppose we sum these quantities. That would give us the average total number of packets waiting in all of the queues of the network. It turns out that Little's formula works in the aggregate as well.[5] Thus, the number of packets waiting and being served in the network can be expressed as γT. Combining the two:

$$T = \frac{1}{\gamma} \sum_{i=1}^{L} \lambda_i T_{ri}$$

[4] This discussion is based on the development in [KLEI76].

[5] In essence, this statement is based on the fact that the sum of the averages is the average of the sums.

To determine the value of T, we need to determine the values of the individual delays, T_{ri}. Because we are assuming each queue can be treated as an independent M/M/1 model, this is easily determined:

$$T_{ri} = \frac{T_{si}}{1 - \rho_i} = \frac{T_{si}}{1 - \lambda_i T_{si}}$$

The service time T_{si} for link i is just the ratio of the average packet length in bits (M) to the data rate on the link in bits per second (R_i). Then:

$$T_{ri} = \frac{\frac{M}{R_i}}{1 - \frac{M\lambda_i}{R_i}} = \frac{M}{R_i - M\lambda_i}$$

Putting all of the elements together, we can calculate the average delay of packets sent through the network:

$$T = \frac{1}{\gamma} \sum_{i=1}^{L} \frac{M\lambda_i}{R_i - M\lambda_i}$$

21.9 OTHER QUEUEING MODELS

In this chapter, we have concentrated on one type of queueing model. There are in fact a number of models, based on two key factors:

- The manner in which blocked items are handled
- The number of traffic sources

When an item arrives at a server and finds that server busy, or arrives at a multiple-server facility and finds all servers busy, that item is said to be blocked. Blocked items can be handled in a number of ways. First, the item can be placed in a queue awaiting a free server. This policy is referred to in the telephone traffic literature as *lost calls delayed*, although in fact the call is not lost. Alternatively, no queue is provided. This in turn leads to two assumptions about the action of the item. The item may wait some random amount of time then try again; this is known as *lost calls cleared*. If the item repeatedly attempts to gain service, with no pause, it is referred to as *lost calls held*. The lost calls delayed model is the most appropriate for most computer and data communications problems. Lost calls cleared is usually the most appropriate in a telephone-switching environment.

The second key element of a traffic model is whether the number of sources is assumed infinite or finite. For an infinite source model, there is assumed to be a fixed arrival rate. For the finite source case, the arrival rate will depend on the number of sources already engaged. Thus, if each of L sources generates arrivals at a rate λ/L, then when the queueing facility is unoccupied, the arrival rate is λ. However, if K sources are in the queueing facility at a particular time, then the instantaneous arrival rate at that time is $\lambda(L - K)/L$. Infinite source models are easier to deal with. The infinite source assumption is reasonable when the number of sources is at least 5–10 times the capacity of the system.

21.10 ESTIMATING MODEL PARAMETERS

To perform a queueing analysis, we need to estimate the values of the input parameters, specifically the mean and standard deviation of the arrival rate and service time. If we are contemplating a new system, these estimates may have to be based on judgment and an assessment of the equipment and work patterns likely to prevail. However, it will often be the case that an existing system is available for examination. For example, a collection of terminals, personal computers, and host computers are interconnected in a building by direct connection and multiplexers, and it is desired to replace the interconnection facility with a LAN. To be able to size the network, it is possible to measure the load currently generated by each device.

Sampling

The measurements that are taken are in the form of samples. A particular parameter, for example, the rate of packets generated by a terminal or the size of packets, is estimated by observing the number of packets generated during a period of time.

The most important quantity to estimate is the mean. For many of the equations in Tables 21.6 and 21.7, this is the only quantity that need be estimated. The estimate is referred to as the sample mean $\overline{X}$ and is calculated as follows:

$$\overline{X} = \frac{1}{N}\sum_{i=1}^{N} X_i$$

where

N = sample size

X_i = ith item in the sample

It is important to note the sample mean is itself a random variable. For example, if you take a sample from some population and calculate the sample mean, and do this a number of times, the calculated values will differ. Thus, we can talk of the mean and standard deviation of the sample mean, or even of the entire probability distribution of the sample mean. To distinguish the concepts, it is common to refer to the probability distribution of the original random variable X as the *underlying distribution*, and the probability distribution of the sample mean $\overline{X}$ as the *sampling distribution of the mean*.

The remarkable thing about the sample mean is that its probability distribution tends to the normal distribution as N increases for virtually all underlying distributions. The assumption of normality breaks down only if N is very small or if the underlying distribution is highly abnormal.

The mean and variance of $\overline{X}$ are as follows:

$$\mathrm{E}[\overline{X}] = \mathrm{E}[X] = \mu$$

$$\mathrm{Var}[\overline{X}] = \frac{\sigma_X^2}{N}$$

Thus, if a sample mean is calculated, its expected value is the same as that of the underlying random variable and the variability of the sample mean around this

expected value decreases as N increases. These characteristics are illustrated in Figure 21.9. The figure shows an underlying exponential distribution with mean value $\mu = 1$. This could be the distribution of service times of a server, or of the interarrival times of a Poisson arrival process. If a sample of size 10 is used to estimate the value of μ, then the expected value is indeed μ, but the actual value could easily be off by as much as 50%. If the sample size is 100, the spread among possible calculated values is considerably tightened, so that we would expect the actual sample mean for any given sample to be much closer to μ.

The sample mean as defined previously can be used directly to estimate the service time of a server. For arrival rate, one can observe the interarrival times for a sequence of N arrivals, calculate the sample mean, then calculate the estimated arrival rate. An equivalent and simpler approach is to use the following estimate:

$$\overline{\lambda} = \frac{N}{T}$$

where N is the number of items observed in a period of time of duration T.

For much of queueing analysis, it is only an estimate of the mean that is required. But for a few important equations, an estimate of the variance of the

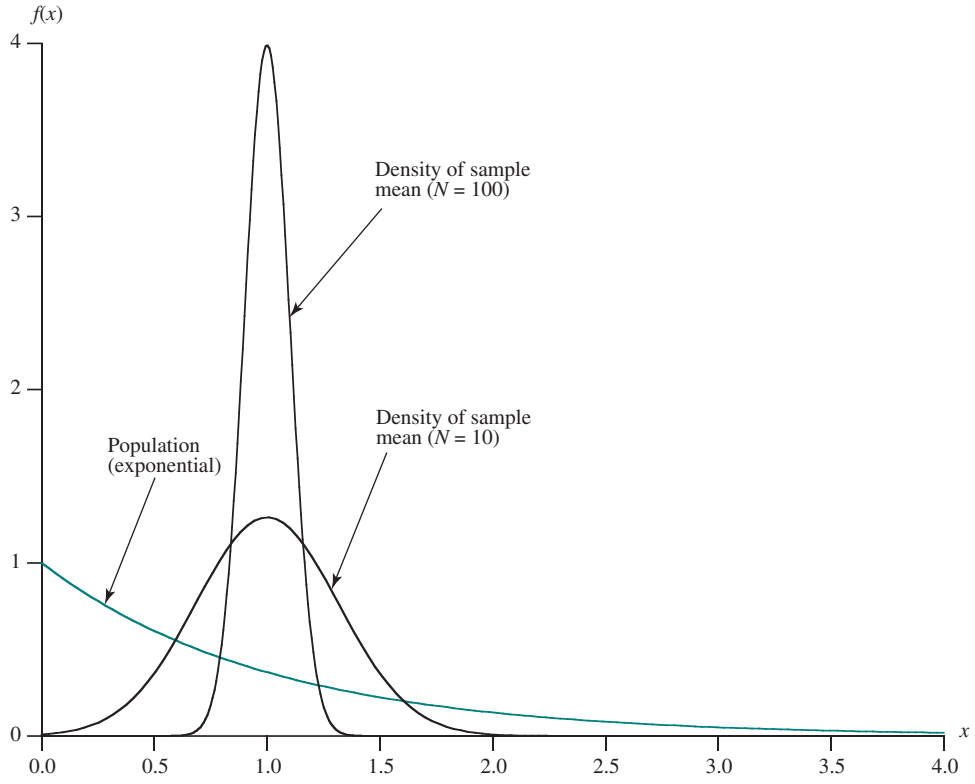


Figure 20.9 Sample Means for an Exponential Population

underlying random variable, σ_X^2, is also needed. The sample variance is calculated as follows:

$$S^2 = \frac{1}{N-1}\sum_{i=1}^{N}(X_i - \overline{X})^2$$

The expected value of S^2 has the desired value:

$$\mathrm{E}[S^2] = \sigma_X^2$$

The variance of S^2 depends on the underlying distribution and is, in general, difficult to calculate. However, as you would expect, the variance of S^2 decreases as N increases.

Table 21.10 summarizes the concepts discussed in this section.

Sampling Errors

When we estimate values such as the mean and standard deviation on the basis of a sample, we leave the realm of probability and enter that of statistics. This is a complex topic that will not be explored here, except to provide a few comments.

The probabilistic nature of our estimated values is a source of error, known as **sampling error**. In general, the greater the size of the sample taken, the smaller the standard deviation of the sample mean or other quantity, and therefore the closer that our estimate is likely to be to the actual value. By making certain reasonable assumptions about the nature of the random variable being tested and the randomness of the sampling procedure, one can in fact determine the probability that a sample mean or sample standard deviation is within a certain distance from the actual mean or standard deviation. This concept is often reported with the results of a sample. For example, it is common for the result of an opinion poll to include a comment such as, "The result is within 5% of the true value with a confidence (probability) of 99%."

There is, however, another source of error, which is less widely appreciated among nonstatisticians: **bias**. For example, if an opinion poll is conducted and only members of a certain socioeconomic group are interviewed, the results are not necessarily representative of the entire population. In a communications context, sampling done during one time of day may not reflect the activity at another time of day. If we are concerned with designing a system that will handle the peak load that is likely to be experienced, then we should observe the traffic during the time of day that is most likely to produce the greatest load.

Table 21.10 Statistical Parameters

	Population	Sample Mean	Sample Variance
Random variable	X	$\overline{X} = \frac{1}{N}\sum_{i=1}^{N} X_i$	$S^2 = \frac{1}{N-1}\sum_{i=1}^{N}(X_i - \overline{X})^2$
Expected value	$\mathrm{E}[X] = \mu$	$\mathrm{E}[\overline{X}] = \mu$	$\mathrm{E}[S^2] = \sigma_X^2$
Variance	$\mathrm{Var}[X] = \mathrm{E}[(X-\mu)^2] = \sigma_X^2$	$\mathrm{Var}[\overline{X}] = \frac{\sigma_X^2}{N}$	

21.11 REFERENCES

KLEI76 Kleinrock, L. *Queueing Systems, Volume II: Computer Applications.* New York: Wiley, 1976.

21.12 PROBLEMS

21.1 Section 21.3 provided an intuitive argument to justify Little's formula. Develop a similar argument to justify the relationship $r = \lambda T_r$.

21.2 Figure 21.3 shows the number of items in a system as a function of time. This can be viewed as the difference between an arrival process and a departure process, of the form $n(t) = a(t) - d(t)$.

a. On one graph, show the functions $a(t)$ and $d(t)$ that produce the $n(t)$ shown in Figure 21.3.

b. Using the graph from (a), develop an intuitive argument to justify Little's formula. *Hint*: Consider the area between the two step functions, computed first by adding vertical rectangles and second by adding horizontal rectangles.

21.3 The owner of a shop observes that on average 18 customers per hour arrive and there are typically 8 customers in the shop. What is the average length of time each customer spends in the shop?

21.4 A simulation program of a multiprocessor system starts running with no jobs in the queue and ends with no jobs in the queue. The simulation program reports the average number of jobs in the system over the simulation run as 12.356, the average arrival rate as 25.6 jobs per minute, and the average delay for a job as 8.34 minutes. Was the simulation correct?

21.5 Section 21.3 provided an intuitive argument to justify the single-server relationship $\rho = \lambda T_s$. Develop a similar argument to justify the multiserver relationship $\rho = \lambda T_s/N$.

21.6 If an M/M/1 queue has arrivals at a rate of two per minute and serves at a rate of four per minute, how many customers are found in the system on average? How many customers are found in service on average?

21.7 What is the utilization of an M/M/1 queue that has four people waiting on average?

21.8 At an ATM machine in a supermarket, the average length of a transaction is two minutes, and on average, customers arrive to use the machine once every five minutes. How long is the average time that a person must spend waiting and using the machine? What is the 90th percentile of residence time? On average, how many people are waiting to use the machine? Assume M/M/1.

21.9 Messages arrive at random to be sent across a communications link with a data rate of 9,600 bps. The link is 70% utilized, and the average message length is 1,000 octets. Determine the average waiting time for constant-length messages and for exponentially distributed length messages.

21.10 Messages of three different sizes flow through a message switch. Seventy percent of the messages take 1 millisecond to serve, 20% take 3 millisecond, and 10% take 10 millisecond. Calculate the average time spent in the switch, and the average number of messages in the switch, when messages arrive at an average rate of:

a. one per 3 milliseconds.

b. one per 4 milliseconds.

c. one per 5 milliseconds.

21.11 Messages arrive at a switching center for a particular outgoing communications line in a Poisson manner with a mean arrival rate of 180 messages per hour. Message length

is distributed exponentially with a mean length of 14,400 characters. Line speed is 9,600 bps.

a. What is the mean waiting time in the switching center?

b. How many messages will be waiting in the switching center for transmission on the average?

21.12 Often inputs to a queueing system are not independent and random, but occur in clusters. Mean waiting delays are greater for this type of arrival pattern than for Poisson arrivals. This problem demonstrates the effect with a simple example. Assume items arrive at a queue in fixed-size batches of M items. The batches have a Poisson arrival distribution with mean rate λ/M, yielding a customer arrival rate of λ. For each item, the service time is T_s, and the standard deviation of service time of σ_{Ts}.

a. If we treat the batches as large-size items, what is the mean and variance of batch service time? What is the mean batch waiting time?

b. What is the mean waiting time for service for an item once its batch begins service? Assume an item may be in any of the M positions in a batch with equal probability. What is the total mean waiting time for an item?

c. Verify the results of (b) by showing that for $M = 1$, the results reduce to the M/G/1 case. How do the results vary for values of $M > 1$?

21.13 Consider a single queue with a constant service time of four seconds and a Poisson input with mean rate of 0.20 items per second.

a. Find the mean and standard deviation of queue size.

b. Find the mean and standard deviation of residence time.

21.14 Consider a frame relay node that is handling a Poisson stream of incoming frames to be transmitted on a particular 1-Mbps outgoing link. The stream consists of two types of frames. Both types of frames have the same exponential distribution of frame length with a mean of 1,000 bits.

a. Assume priorities are not used. The combined arrival rate of frame of both types is 800 frames per second. What is the mean residence time (T_r) for all frames?

b. Now assume the two types are assigned different priorities, with the arrival rate of type 1 of 200 frames per second and the arrival rate of type 2 of 600 frames per second. Calculate the mean residence time for type 1, type 2, and overall.

c. Repeat (b) for $\lambda_1 = \lambda_2 = 400$ frames per second.

d. Repeat (b) for $\lambda_1 = 600$ frames per second and $\lambda_2 = 200$ frames per second.

21.15 The Multilink Protocol (MLP) is part of X.25; a similar facility is used in IBM's System Network Architecture (SNA). With MLP, a set of data links exists between two nodes and is used as a pooled resource for transmitting packets, regardless of virtual circuit number. When a packet is presented to MLP for transmission, any available link can be chosen for the job. For example, if two LANs at different sites are connected by a pair of bridges, there may be multiple point-to-point links between the bridges to increase throughput and availability.

The MLP approach requires extra processing and frame overhead compared to a simple link protocol. A special MLP header is necessary for the protocol. An alternative is to assign each of the arriving packets to the queue for a single outgoing link in round-robin fashion. This would simplify processing, but what kind of effect would it have on performance?

Let us consider a concrete example. Suppose there are five 9,600-bps links connecting two nodes, the average packet size is 100 octets with an exponential distribution, and packets arrive at a rate of 48 per second.

a. For a single-server design, calculate ρ and T_r.

b. For a multiserver design, it can be calculated that the Erlang C function has a value of 0.554. Determine T_r.

21.16 A supplement to the X.25 packet-switching standard is a set of standards for a packet assembler-disassembler (PAD), defined in standards X.3, X.28, and X.29. A PAD is used to connect asynchronous terminals to a packet-switching network. Each terminal attached to a PAD sends characters one at a time. These are buffered in the PAD then assembled into an X.25 packet that is transmitted to the packet-switching network. The buffer length is equal to the maximum data field size for an X.25 packet. A packet is formed from assembled characters and transmitted whenever the buffer is full, a special control character such as a carriage return is received, or when a timeout occurs. For this problem, we ignore the last two conditions. Figure 21.10 illustrates the queueing model for the PAD. The first queue models the delay for characters waiting to be put into a packet; this queue is completely emptied when it is filled. The second queue models the delay waiting to transmit packets. Use the following notation:

λ = Poisson input rate of characters from each termina.l
C = Rate of transmission on the output channel in characters per second.
M = Number of data characters in a packet.
H = Number of overhead characters in a packet.
K = Number of terminals.

a. Determine the average waiting time for a character in the input queue.

b. Determine the average waiting time for a packet in the output queue.

c. Determine the average time spent by a character from when it leaves the terminal to when it leaves the PAD. Plot the result as a function of normalized load.

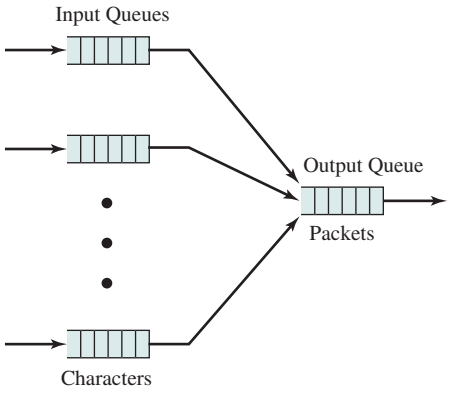


Figure 21.10 Queueing Model for a Packet Assembler/Disassembler (PAD)

21.17 A fraction P of the traffic from a single exponential server is fed back into the input as shown in Figure 21.11. In the figure, Λ denotes the system throughput, which is the output rate from the server.

a. Determine the system throughput and the server utilization and the mean residence time for one pass through the server.

b. Determine the mean number of passes that an item makes through the system and the mean total time spent in the system.

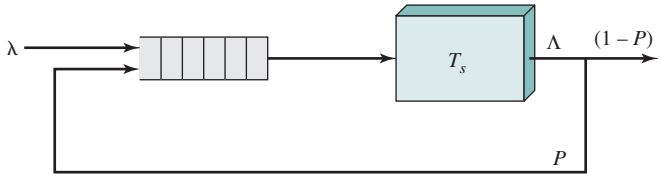


Figure 21.11 Feedback Queue

PROGRAMMING PROJECT ONE

DEVELOPING A SHELL

The Shell or Command Line Interpreter is the fundamental User interface to an operating system. Your first project is to write a simple shell—`myshell`—that has the following properties:

1. The shell must support the following internal commands:
   - i. `cd <directory>`—Change the current default directory to `<directory>`. If the `<directory>` argument is not present, report the current directory. If the directory does not exist, an appropriate error should be reported. This command should also change the `PWD` environment variable.
   - ii. `clr`—Clear the screen.
   - iii. `dir <directory>`—List the contents of directory `<directory>`.
   - iv. `environ`—List all the environment strings.
   - v. `echo <comment>`—Display `<comment>` on the display followed by a new line (multiple spaces/tabs may be reduced to a single space).
   - vi. `help`—Display the user manual using the `more` filter.
   - vii. `pause`—Pause operation of the shell until "`Enter`" is pressed.
   - viii. `quit`—Quit the shell.
   - ix. The shell environment should contain `shell=<pathname>/myshell` where `<pathname>/myshell` is the full path for the shell executable (not a hardwired path back to your directory, but the one from which it was executed).
2. All other command line input is interpreted as program invocation, which should be done by the shell `fork`ing and `exec`ing the programs as its own child processes. The programs should be executed with an environment that contains the entry: `parent=<pathname>/myshell` where `<pathname>/myshell` is as described in 1.ix above.
3. The shell must be able to take its command line input from a file. That is, if the shell is invoked with a command line argument:

   ```
   myshell batchfile
   ```

   then `batchfile` is assumed to contain a set of command lines for the shell to process. When the end-of-file is reached, the shell should exit. Obviously, if the shell is invoked without a command line argument, it solicits input from the user via a prompt on the display.
4. The shell must support I/O redirection on either or both *stdin* and/or *stdout*. That is, the command line

   ```
   programname arg1 arg2 < inputfile > outputfile
   ```

   will execute the program `programname` with arguments `arg1` and `arg2`, the *stdin FILE stream* replaced by `inputfile` and the *stdout FILE stream* replaced by `outputfile`.

`stdout` redirection should also be possible for the internal commands `dir`, `environ`, `echo`, and `help`.

With output redirection, if the redirection character is > then the `outputfile` is created if it does not exist, and truncated if it does. If the redirection token is >> then `outputfile` is created if it does not exist, and appended to if it does.

5. The shell must support background execution of programs. An ampersand (`&`) at the end of the command line indicates that the shell should return to the command line prompt immediately after launching that program.
6. The command line prompt must contain the pathname of the current directory.

Note: You can assume all command line arguments (including the redirection symbols, <, > & >> and the background execution symbol, `&`) will be delimited from other command line arguments by white space—one or more spaces and/or tabs (see the command line in 4. above).

PROJECT REQUIREMENTS

1. Design a simple command line shell that satisfies the above criteria and implement it on the specified UNIX platform.
2. Write a simple manual describing how to use the shell. The manual should contain enough detail for a beginner to UNIX to use it. For example, you should explain the concepts of I/O redirection, the program environment, and background program execution. The manual MUST be named `readme` and must be a simple text document capable of being read by a standard Text Editor.

   For an example of the sort of depth and type of description required, you should have a look at the online manuals for `csh` and `tcsh` (`man csh`, `man tcsh`). These shells obviously have much more functionality than yours and thus, your manuals don't have to be quite so large.

   You should NOT include building instructions, included file lists, or source code—we can find that out from the other files you submit. This should be an Operator's manual not a Developer's manual.
3. The source code **MUST** be extensively commented and appropriately structured to allow your peers to understand and easily maintain the code. Properly commented and laid out code is much easier to interpret, and it is in your interests to ensure the person marking your project is able to understand your coding without having to perform mental gymnastics!
4. Details of submission procedures will be supplied well before the deadline.
5. The submission should contain only source code file(s), include file(s), a `makefile` (all lowercase please), and the `readme` file (all lowercase, please). No executable program should be included. The person marking your project will be automatically rebuilding your shell program from the source code provided. If the submitted code does not compile, it cannot be marked!

6. The `makefile` (all lowercase, please) **MUST** generate the binary file `myshell` (all lowercase please). A sample `makefile` would be

```
# Joe Citizen, s1234567 - Operating Systems Project 1
# CompLab1/01 tutor: Fred Bloggs
myshell: myshell.c utility.c myshell.h
    gcc -Wall myshell.c utility.c -o myshell
```

   The program `myshell` is then generated by just typing `make` at the command line prompt.

   Note: The fourth line in the above `makefile` **MUST** begin with a tab.

7. In the instance shown above, the files in the submitted directory would be:

```
makefile
myshell.c
utility.c
myshell.h
readme
```

SUBMISSION

A `makefile` is required. All files in your submission will be copied to the same directory, therefore, do not include any paths in your `makefile`. The `makefile` should include all dependencies that build your program. If a library is included, your `makefile` should also build the library.

Do not hand in any binary or object code files. All that is required is your source code, a `makefile`, and a `readme` file. Test your project by copying the source code only into an empty directory then compile it by entering the command `make`.

We shall be using a shell script that copies your files to a test directory, deletes any preexisting `myshell`, `*.a`, and/or `*.o` files, performs a `make`, copies a set of test files to the test directory, and then exercises your shell with a standard set of test scripts through *stdin* and command line arguments. If this sequence fails due to wrong names, wrong case for names, wrong version of source code that fails to compile, nonexistence of files, and so on, then the marking sequence will also stop. In this instance, the only marks that can be awarded will be for the tests completed at that point, and the source code and manual.

REQUIRED DOCUMENTATION

Your source code will be assessed and marked as well as the `readme` manual. Commenting is definitely required in your source code. The user manual can be presented in a format of your choice (within the limitations of being displayable by a simple Text Editor). Again, the manual should contain enough detail for a beginner to UNIX to use the shell. For example, you should explain the concepts of I/O redirection, the program environment, and background program execution. The manual **MUST** be named `readme` (all lowercase, please, and **NO** `.txt` extension).

PROGRAMMING PROJECT TWO

THE HOST DISPATCHER SHELL

The Hypothetical Operating System Testbed (HOST) is a multiprogramming system with a four-level priority process dispatcher operating within the constraints of finite available resources.

FOUR-LEVEL PRIORITY DISPATCHER

The dispatcher operates at four priority levels:

1. Real-Time processes must be run immediately on a first-come-first-served (FCFS) basis, preempting any other processes running with lower priority. These processes are run until completion.
2. Normal user processes are run on a three-level feedback dispatcher (see Figure PP2.1). The basic timing quantum of the dispatcher is one second. This is also the value for the time quantum of the feedback scheduler.

The dispatcher needs to maintain two submission queues—Real-Time and User priority—fed from the job dispatch list. The dispatch list is examined at every dispatcher tick and jobs that "have arrived" are transferred to the appropriate submission queue. The submission queues are then examined; any Real-Time jobs are run to completion, preempting any other jobs currently running.

The Real-Time priority job queue must be empty before the lower-priority feedback dispatcher is reactivated. Any User priority jobs in the User job queue that can run within available resources (memory and I/O devices) are transferred to the appropriate priority queue. Normal operation of a feedback queue will accept all

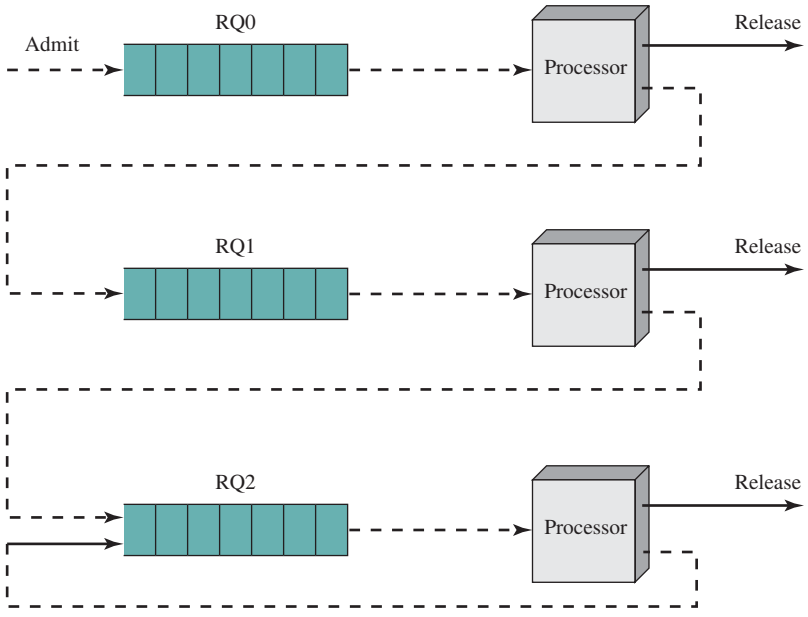


Figure PP2.1

Figure PP2.2

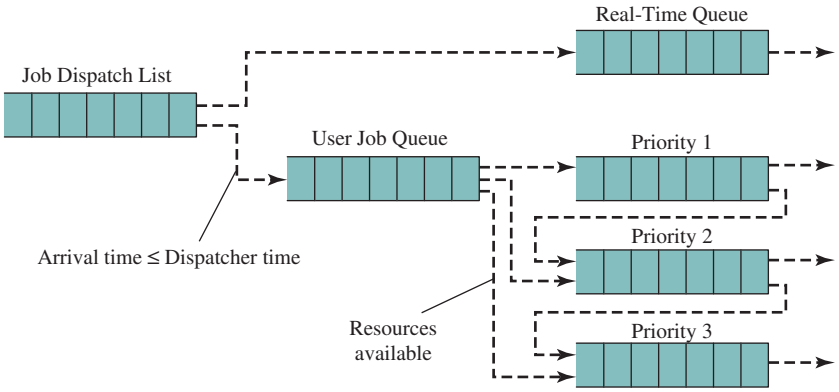


Figure PP2.3

jobs at the highest-priority level and degrade the priority after each completed time quantum. However, this dispatcher has the ability to accept jobs at a lower priority, inserting them in the appropriate queue. This enables the dispatcher to emulate a simple round-robin dispatcher (see Figure PP2.2) if all jobs are accepted at the lowest priority.

When all "ready" higher-priority jobs have been completed, the feedback dispatcher resumes by starting or resuming the process at the head of the highest-priority nonempty queue. At the next tick the current job is suspended (or terminated and its resources released) if there are any other jobs "ready" of an equal or higher priority.

The logic flow should be as shown in Figure PP2.3 (and as discussed subsequently in this project assignment).

RESOURCE CONSTRAINTS

The HOST has the following resources:

- 2 Printers
- 1 Scanner
- 1 Modem
- 2 CD drives
- 1024 Mbytes of memory available for processes

Low-priority processes can use any or all of these resources, but the HOST dispatcher is notified of which resources the process will use when the process is submitted. The dispatcher ensures that each requested resource is solely available to that process throughout its lifetime in the "ready-to-run" dispatch queues: from the initial transfer from the job queue to the Priority 1–3 queues through to process completion, including intervening idle time quanta.

Real-Time processes will not need any I/O resources (Printer, Scanner, Modem, and CD), but will obviously require memory allocation—this memory requirement will always be 64 Mbytes or less for Real-Time jobs.

MEMORY ALLOCATION

For each process, a **contiguous** block of memory must be assigned. The memory block must remain assigned to the process for the lifetime of the process.

Enough contiguous spare memory must be left so the Real-Time processes are not blocked from execution—64 Mbytes for a running Real-Time job, leaving 960 Mbytes to be shared among "active" User jobs.

The HOST hardware MMU cannot support virtual memory, so no swapping of memory to disk is possible. Neither is it a paged system.

Within these constraints, any suitable variable partition memory allocation scheme (First Fit, Next Fit, Best Fit, Worst Fit, Buddy, and so on) may be used.

PROCESSES

Processes on HOST are simulated by the dispatcher creating a new process for each dispatched process. This process is a generic process (supplied as `process`—source: `sigtrap.c`) that can be used for any priority process. It actually runs itself at very low priority, sleeping for one-second periods and displaying the following:

1. A message displaying the process ID when the process starts;
2. A regular message every second the process is executed, and;
3. A message when the process is Suspended, Continued, or Terminated.

The process will terminate of its own accord after 20 seconds if it is not terminated by your dispatcher. The process prints out using a randomly generated color scheme for each unique process, so individual "slices" of processes can be easily distinguishable. Use this process rather than your own.

The life cycle of a process is as follows:

1. The process is submitted to the dispatcher input queues via an initial process list that designates the arrival time, priority, processor time required (in seconds), memory block size, and other resources requested.
2. A process is "ready-to-run" when it has "arrived" and all required resources are available.
3. Any pending Real-Time jobs are submitted for execution on a FCFS basis.

4. If enough resources and memory are available for a lower-priority User process, the process is transferred to the appropriate priority queue within the feedback dispatcher unit, and the remaining resource indicators (memory list and I/O devices) are updated.
5. When a job is started (`fork` and `exec("process",...)`), the dispatcher will display the job parameters (Process ID, priority, processor time remaining (in seconds), memory location and block size, and resources requested) before performing the `exec`.
6. A Real-Time process is allowed to run until its time has expired when the dispatcher kills it by sending a `SIGINT` signal to it.
7. A low-priority User job is allowed to run for one dispatcher tick (one second) before it is suspended (`SIGTSTP`) or terminated (`SIGINT`) if its time has expired. If suspended, its priority level is lowered (if possible) and it is requeued on the appropriate priority queue as shown in Figures P2.1 and P2.3. To retain synchronization of output between your dispatcher and the child process, your dispatcher should wait for the process to respond to a `SIGTSTP` or `SIGINT` signal before continuing (`waitpid(p->pid, &status, WUNTRACED)`). To match the performance sequence indicated in the comparison of scheduling policies (see Figure 9.5), the User job should not be suspended and moved to a lower-priority level unless another process is waiting to be (re)started.
8. Provided no higher-priority Real-Time jobs are pending in the submission queue, the highest-priority pending process in the feedback queues is started or restarted (`SIGCONT`).
9. When a process is terminated, the resources it used are returned to the dispatcher for reallocation to further processes.
10. When there are no more processes in the dispatch list—the input queues and the feedback queues—the dispatcher exits.

DISPATCH LIST

The Dispatch List is the list of processes to be processed by the dispatcher. The list is contained in a text file that is specified on the command line. That is,

```
>hostd dispatchlist
```

Each line of the list describes one process with the following data as a *"comma-space"* delimited list:

```
<arrival time>, <priority>, <processor time>, <mbytes>,
<#printers>, <#scanners>, <#modems>, <#CDs>
```

Thus,

```
12, 0, 1, 64, 0, 0, 0, 0
12, 1, 2, 128, 1, 0, 0, 1
13, 3, 6, 128, 1, 0, 1, 2
```

would indicate the following:

1st Job:	Arrival at time 12, priority 0 (Real-Time), requiring 1 second of processor time and 64 Mbytes of memory—no I/O resources required.
2nd Job:	Arrival at time 12, priority 1 (high-priority User job), requiring 2 seconds of processor time, 128 Mbytes of memory, 1 printer, and 1 CD drive.
3rd Job:	Arrival at time 13, priority 3 (lowest-priority User job), requiring 6 seconds of processor time, 128 Mbytes of memory, 1 printer, 1 modem, and 2 CD drives.

The submission text file can be of any length, containing up to 1000 jobs. It will be terminated with an end-of-line followed by an end-of-file marker.

Dispatcher input lists to test the operation of the individual features of the dispatcher are described subsequently in this project assignment. It should be noted that these lists will almost certainly form the basis of tests that will be applied to your dispatcher during marking. Operation as described in the exercises will be expected.

Obviously, your submitted dispatcher will be tested with more complex combinations as well!

A fully functional working example of the dispatcher will be presented during the course. If in any doubt as to the manner of operation or format of output, you should refer to this program to observe how your dispatcher is expected to operate.

PROJECT REQUIREMENTS

1. Design a dispatcher that satisfies the above criteria. In a formal design document,

   a. Describe and discuss what memory allocation algorithms you could have used and justify your final design choice.

   b. Describe and discuss the structures used by the dispatcher for queueing, dispatching, and allocating memory and other resources.

   c. Describe and justify the overall structure of your program, describing the various modules and major functions (descriptions of the function "interfaces" are expected).

   d. Discuss why such a multilevel dispatching scheme would be used, comparing it with schemes used by "real" operating systems. Outline shortcomings in such a scheme, suggesting possible improvements. Include the memory and resource allocation schemes in your discussions.

   The formal design document is expected to have in-depth discussions, descriptions, and arguments. The design document is to be submitted separately as a physical paper document. The design document should NOT include any source code.

2. Implement the dispatcher using the C language.
3. The source code MUST be extensively commented and appropriately structured to allow your peers to understand and easily maintain the code. Properly commented and laid out code is much easier to interpret, and it is in your interests to ensure the person marking your project is able to understand your coding without having to perform mental gymnastics.
4. Details of submission procedures will be supplied well before the deadline.
5. The submission should contain only source code file(s), include file(s), and a `makefile`. No executable program should be included. The marker will be automatically rebuilding your program from the source code provided. If the submitted code does not compile, it cannot be marked.
6. The `makefile` should generate the binary executable file `hostd` (all lowercase please). A sample `makefile` would be as follows:

```
# Joe Citizen, s1234567 - Operating Systems Project 2
# CompLab1/01 tutor: Fred Bloggs
hostd: hostd.c utility.c hostd.h
gcc hostd.c utility.c -o hostd
```

The program `hostd` is then generated by typing `make` at the command line prompt. Note: The fourth line in the above `makefile` **MUST** begin with a `tab`.

DELIVERABLES

1. Source code file(s), include file(s), and a `makefile`.
2. The design document as outlined in Project Requirements section 1 above.

SUBMISSION OF CODE

A `makefile` is required. All files will be copied to the same directory; therefore, *do not include any paths in your makefile*. The `makefile` should include all dependencies that build your program. If a library is included, your `makefile` should also build the library.

Do not submit any binary or object code files. All that is required is your source code and a `makefile`. Test your project by copying the source code only into an *empty* directory then compile it with your `makefile`.

The marker will be using a shell script that copies your files to a test directory, performs a `make`, then exercises your dispatcher with a standard set of test files. If this sequence fails due to wrong names, wrong case for names, wrong version of source code that fails to compile, nonexistence of files, etc., then the marking sequence will also stop. In this instance, the only further marks that can be awarded will be for the source code and design document.

APPENDIX C

TOPICS IN CONCURRENCY

This appendix provides additional details to supplement Chapter 1.

C.1 PROCESSOR REGISTERS

A processor includes a set of registers that provide memory that is faster and smaller than main memory. Processor registers serve two functions:

- **User-visible registers:** Enable the machine or assembly language programmer to minimize main memory references by optimizing register use. For high-level languages, an optimizing compiler will attempt to make intelligent choices of which variables to assign to registers and which to main memory locations. Some high-level languages such as C allow the programmer to suggest to the compiler which variables should be held in registers.
- **Control and status registers:** Used by the processor to control the operation of the processor, and by privileged OS routines to control the execution of programs.

There is not a clean separation of registers into these two categories. For example, on some processors, the program counter is user visible, but on many it is not. For purposes of the following discussion, however, it is convenient to use these categories.

User-Visible Registers

A user-visible register may be referenced by means of the machine language that the processor executes and is generally available to all programs, including application programs as well as system programs. Types of registers that are typically available are data, address, and condition code registers.

Data registers can be assigned to a variety of functions by the programmer. In some cases, they are general purpose in nature and can be used with any machine instruction that performs operations on data. Often, however, there are restrictions. For example, there may be dedicated registers for floating-point operations, and others for integer operations.

Address registers contain main memory addresses of data and instructions, or they contain a portion of the address that is used in the calculation of the complete or effective address. These registers may themselves be general purpose, or may be devoted to a particular way, or mode, of addressing memory. Examples include the following:

- **Index register:** Indexed addressing is a common mode of addressing that involves adding an index to a base value to get the effective address.
- **Segment pointer:** With segmented addressing, memory is divided into segments, which are variable-length blocks of words.[1] A memory reference consists of a reference to a particular segment and an offset within the segment;

[1] There is no universal definition of the term *word*. In general, a **word** is an ordered set of bytes or bits that is the normal unit in which information may be stored, transmitted, or operated on within a given computer. Typically, if a processor has a fixed-length instruction set, then the instruction length equals the word length.

this mode of addressing is important in our discussion of memory management in Chapter 7. In this mode of addressing, a register is used to hold the base address (starting location) of the segment. There may be multiple registers; for example, one for the OS (i.e., when OS code is executing on the processor) and one for the currently executing application.

- **Stack pointer:** If there is user-visible stack[2] addressing, then there is a dedicated register that points to the top of the stack. This allows the use of instructions that contain no address field, such as push and pop.

For some processors, a procedure call will result in automatic saving of all user-visible registers, to be restored on return. Saving and restoring is performed by the processor as part of the execution of the call and return instructions. This allows each procedure to use these registers independently. On other processors, the programmer must save the contents of the relevant user-visible registers prior to a procedure call, by including instructions for this purpose in the program. Thus, the saving and restoring functions may be performed in either hardware or software, depending on the processor.

Control and Status Registers

A variety of processor registers are employed to control the operation of the processor. On most processors, most of these are not visible to the user. Some of them may be accessible by machine instructions executed in what is referred to as a control or kernel mode.

Of course, different processors will have different register organizations and use different terminology. We provide here a reasonably complete list of register types, with a brief description. In addition to the MAR, MBR, I/OAR, and I/OBR registers mentioned in Chapter 1 (see Figure 1.1), the following are essential to instruction execution:

- **Program counter (PC):** Contains the address of the next instruction to be fetched
- **Instruction register (IR):** Contains the instruction most recently fetched

All processor designs also include a register or set of registers, often known as the program status word (PSW) that contains status information. The PSW typically contains condition codes plus other status information, such as an interrupt enable/disable bit and a kernel/user mode bit.

Condition codes (also referred to as *flags*) are bits typically set by the processor hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set following the execution of the arithmetic instruction. The condition code may subsequently be tested as part of a conditional branch operation. Condition code bits are collected into one or more registers. Usually, they form part of a control register. Generally, machine instructions

[2]A stack is located in main memory and is a sequential set of locations that are referenced similarly to a physical stack of papers, by putting on and taking away from the top. See Appendix P for a discussion of stack processing.

allow these bits to be read by implicit reference, but they cannot be altered by explicit reference because they are intended for feedback regarding the results of instruction execution.

In processors with multiple types of interrupts, a set of interrupt registers may be provided, with one pointer to each interrupt-handling routine. If a stack is used to implement certain functions (e.g., procedure call), then a stack pointer is needed (see Appendix 1B). Memory management hardware, discussed in Chapter 7, requires dedicated registers. Finally, registers may be used in the control of I/O operations.

A number of factors go into the design of the control and status register organization. One key issue is OS support. Certain types of control information are of specific utility to the OS. If the processor designer has a functional understanding of the OS to be used, then the register organization can be designed to provide hardware support for particular features such as memory protection and switching between user programs.

Another key design decision is the allocation of control information between registers and memory. It is common to dedicate the first (lowest) few hundred or thousand words of memory for control purposes. The designer must decide how much control information should be in more expensive, faster registers and how much in less expensive, slower main memory.

C.2 INSTRUCTION EXECUTION FOR I/O FUNCTIONS

This section supplements the information in Section 1.3.

Data can be exchanged directly between an I/O module (e.g., a disk controller) and the processor. Just as the processor can initiate a read or write with memory, specifying the address of a memory location, the processor can also read data from or write data to an I/O module. In this latter case, the processor identifies a specific device that is controlled by a particular I/O module. Thus, an instruction sequence similar in form to that of Figure 1.4 could occur, with I/O instructions rather than memory-referencing instructions.

In some cases, it is desirable to allow I/O exchanges to occur directly with main memory to relieve the processor of the I/O task. In such a case, the processor grants to an I/O module the authority to read from or write to memory, so the I/O-memory transfer can occur without tying up the processor. During such a transfer, the I/O module issues read or write commands to memory, relieving the processor of responsibility for the exchange. This operation, known as direct memory access (DMA), is examined in Section 1.7.

C.3 I/O COMMUNICATION TECHNIQUES

Three techniques are possible for I/O operations:

- Programmed I/O
- Interrupt-driven I/O
- Direct memory access (DMA)

Programmed I/O

When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. In the case of programmed I/O, the I/O module performs the requested action then sets the appropriate bits in the I/O status register, but takes no further action to alert the processor. In particular, it does not interrupt the processor. Thus, after the I/O instruction is invoked, the processor must take some active role in determining when the I/O instruction is completed. For this purpose, the processor periodically checks the status of the I/O module until it finds that the operation is complete.

With this technique, the processor is responsible for extracting data from main memory for output, and storing data in main memory for input. I/O software is written in such a way that the processor executes instructions that give it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data. Thus, the instruction set includes I/O instructions in the following categories:

- **Control:** Used to activate an external device and tell it what to do. For example, a magnetic-tape unit may be instructed to rewind or to move forward one record.
- **Status:** Used to test various status conditions associated with an I/O module and its peripherals.
- **Transfer:** Used to read and/or write data between processor registers and external devices.

Figure C.1a gives an example of the use of programmed I/O to read in a block of data from an external device (e.g., a record from tape) into memory. Data are read in one word (e.g., 16 bits) at a time. For each word that is read in, the processor must remain in a status-checking loop until it determines that the word is available in the I/O module's data register. This flowchart highlights the main disadvantage of this technique: It is a time-consuming process that keeps the processor busy needlessly.

Interrupt-Driven I/O

With programmed I/O, the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of more data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the performance level of the entire system is severely degraded.

An alternative is for the processor to issue an I/O command to a module then go on to do some other useful work. The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer, as before, and resumes its former processing.

Let us consider how this works, first from the point of view of the I/O module. For input, the I/O module receives a READ command from the processor. The I/O module then proceeds to read data in from an associated peripheral. Once the data are in the module's data register, the module signals an interrupt to the processor over a control line. The module then waits until its data are requested by the

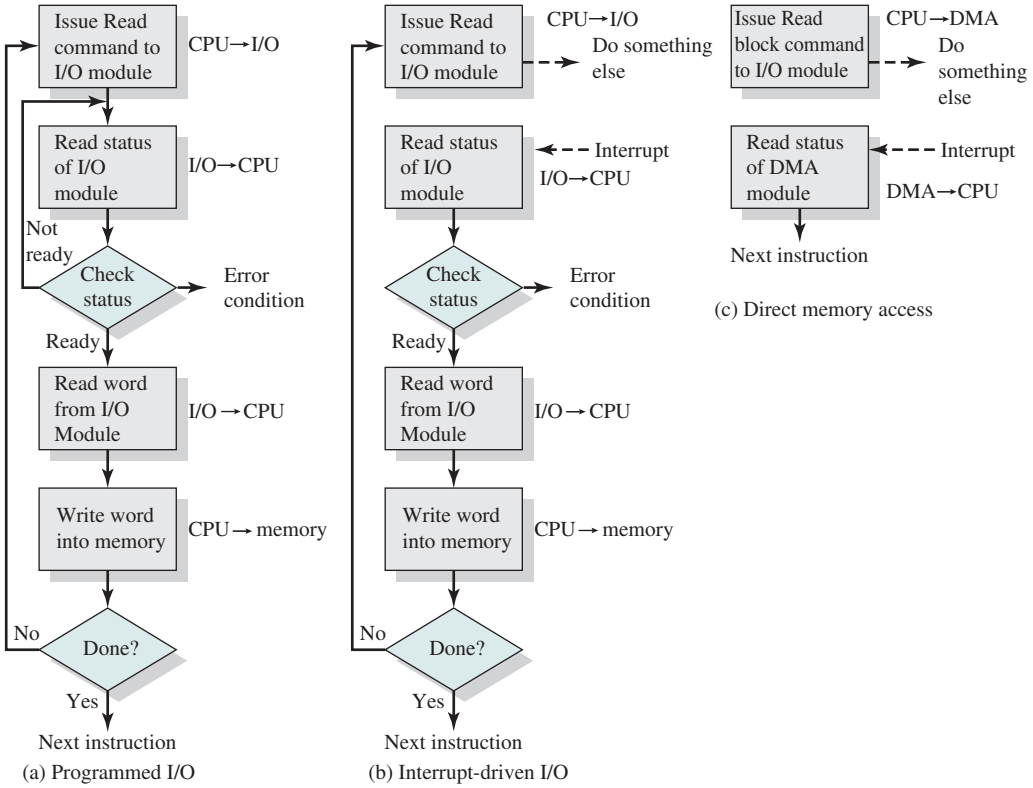


Figure C.1 Three Techniques for Input of a Block of Data

processor. When the request is made, the module places its data on the data bus and is then ready for another I/O operation.

From the processor's point of view, the action for input is as follows. The processor issues a READ command. It then saves the context (e.g., program counter and processor registers) of the current program, and goes off and does something else (e.g., the processor may be working on several different programs at the same time). At the end of each instruction cycle, the processor checks for interrupts (see Figure 1.7). When the interrupt from the I/O module occurs, the processor saves the context of the program it is currently executing and begins to execute an interrupt-handling program that processes the interrupt. In this case, the processor reads the word of data from the I/O module and stores it in memory. It then restores the context of the program that had issued the I/O command (or some other program) and resumes execution.

Figure C.1b shows the use of interrupt-driven I/O for reading in a block of data. Interrupt-driven I/O is more efficient than programmed I/O because it eliminates needless waiting. However, interrupt-driven I/O still consumes a lot of processor time, because every word of data that goes from memory to I/O module, or from I/O module to memory, must pass through the processor.

Almost invariably, there will be multiple I/O modules in a computer system, so mechanisms are needed to enable the processor to determine which device caused the interrupt and to decide, in the case of multiple interrupts, which one to handle first. In some systems, there are multiple interrupt lines, so that each I/O module signals on a different line. Each line will have a different priority. Alternatively, there can be a single interrupt line, but additional lines are used to hold a device address. Again, different devices are assigned different priorities.

Direct Memory Access (DMA)

Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the processor. Thus, both of these forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.
2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA). The DMA function can be performed by a separate module on the system bus or it can be incorporated into an I/O module. In either case, the technique works as follows. When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending the following information to the DMA module:

- Whether a read or write is requested
- The address of the I/O device involved
- The starting location in memory to read data from or write data to
- The number of words to be read or written

The processor then continues with other work. It has delegated this I/O operation to the DMA module, and that module will take care of it. The DMA module transfers the entire block of data, one word at a time, directly to or from memory without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus the processor is involved only at the beginning and end of the transfer (see Figure C.1c).

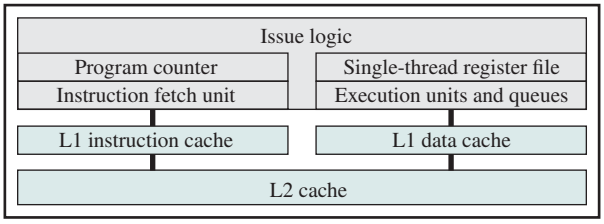
The DMA module needs to take control of the bus to transfer data to and from memory. Because of this competition for bus usage, there may be times when the processor needs the bus and must wait for the DMA module. Note this is not an interrupt; the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle (the time it takes to transfer one word across the bus). The overall effect is to cause the processor to execute more slowly during a DMA transfer when processor access to the bus is required. Nevertheless, for a multiple-word I/O transfer, DMA is far more efficient than interrupt-driven or programmed I/O.

C.4 HARDWARE PERFORMANCE ISSUES FOR MULTICORE

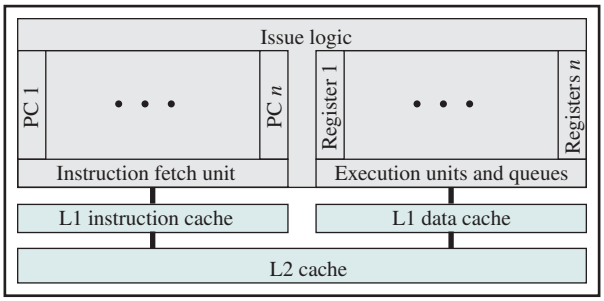
Microprocessor systems have experienced a steady, exponential increase in execution performance for decades. This increase is due partly to refinements in the organization of the processor on the chip, and partly to the increase in the clock frequency.

Increase in Parallelism

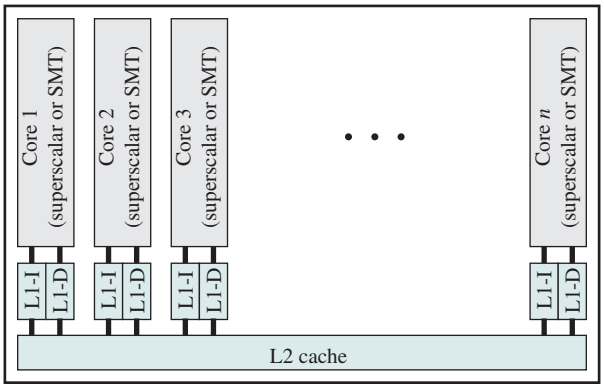
The organizational changes in processor design have primarily been focused on increasing instruction-level parallelism, so more work could be done in each clock cycle. These changes include, in chronological order (see Figure C.2):



(a) Superscalar



(b) Simultaneous multithreading



(c) Multicore

Figure C.2 Alternative Chip Organizations

- **Pipelining:** Individual instructions are executed through a pipeline of stages so while one instruction is executing in one stage of the pipeline, another instruction is executing in another stage of the pipeline.
- **Superscalar:** Multiple pipelines are constructed by replicating execution resources. This enables parallel execution of instructions in parallel pipelines, so long as hazards are avoided.
- **Simultaneous multithreading (SMT):** Register banks are replicated so multiple threads can share the use of pipeline resources.

For each of these innovations, designers have over the years attempted to increase the performance of the system by adding complexity. In the case of pipelining, simple three-stage pipelines were replaced by pipelines with five stages, then many more stages, with some implementations having over a dozen stages. There is a practical limit to how far this trend can be taken, because with more stages, there is the need for more logic, more interconnections, and more control signals. With superscalar organization, performance increases can be achieved by increasing the number of parallel pipelines. Again, there are diminishing returns as the number of pipelines increases. More logic is required to manage hazards and to stage instruction resources. Eventually, a single thread of execution reaches the point where hazards and resource dependencies prevent the full use of the multiple pipelines available. This same point of diminishing returns is reached with SMT, as the complexity of managing multiple threads over a set of pipelines limits the number of threads and number of pipelines that can be effectively utilized.

There is a related set of problems dealing with the design and fabrication of the computer chip. The increase in complexity to deal with all of the logical issues related to very long pipelines, multiple superscalar pipelines, and multiple SMT register banks means that increasing amounts of the chip area is occupied with coordinating and signal transfer logic. This increases the difficulty of designing, fabricating, and debugging the chips. The increasingly difficult engineering challenge related to processor logic is one of the reasons that an increasing fraction of the processor chip is devoted to the simpler memory logic. Power issues, discussed next, provide another reason.

Power Consumption

To maintain the trend of higher performance as the number of transistors per chip rise, designers have resorted to more elaborate processor designs (pipelining, superscalar, and SMT) and to high clock frequencies. Unfortunately, power requirements have grown exponentially as chip density and clock frequency have risen.

One way to control power density is to use more of the chip area for cache memory. Memory transistors are smaller and have a power density an order of magnitude lower than that of logic (see Figure C.3). The percentage of the chip area devoted to memory has grown to exceed 50% as the chip transistor density has increased.

How to use all those logic transistors is a key design issue. As discussed earlier in this section, there are limits to the effective use of such techniques as superscalar and SMT. In general terms, the experience of recent decades has been encapsulated

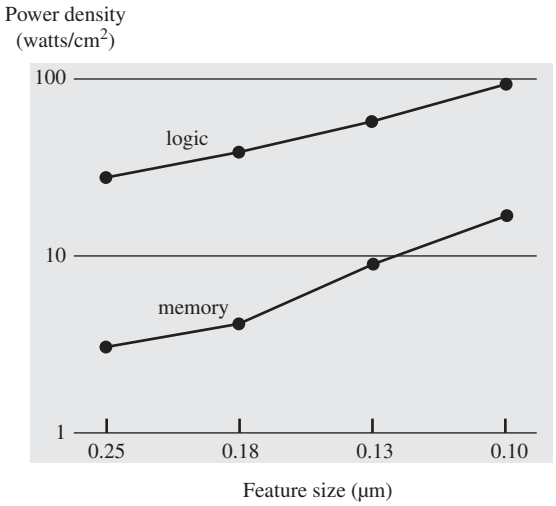


Figure C.3 Power and Memory Considerations

in a rule of thumb known as **Pollack's rule** [POLL99], which states that performance increase is roughly proportional to square root of increase in complexity. In other words, if you double the logic in a processor core, then it delivers only 40% more performance. In principle, the use of multiple cores has the potential to provide near-linear performance improvement with the increase in the number of cores.

Power considerations provide another motive for moving toward a multicore organization. Because the chip has such a huge amount of cache memory, it becomes unlikely that any one thread of execution can effectively use all that memory. Even with SMT, you are multithreading in a relatively limited fashion and cannot therefore fully exploit a gigantic cache, whereas a number of relatively independent threads or processes has a greater opportunity to take full advantage of the cache memory.

C.5 REFERENCE

POLL99 Pollack, F. "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (keynote address)." *Proceedings of the 32nd annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

APPENDIX D

OBJECT-ORIENTED DESIGN

Windows and several other contemporary operating systems rely heavily on object-oriented design principles. This appendix provides a brief overview of the main concepts of object-oriented design.

D.1 MOTIVATION

Object-oriented concepts have become quite popular in the area of computer programming, with the promise of interchangeable, reusable, easily updated, and easily interconnected software parts. More recently, database designers have begun to appreciate the advantages of an object orientation, with the result that object-oriented database management systems (OODBMS) are beginning to appear. Operating systems designers have also recognized the benefits of the object-oriented approach.

Object-oriented programming and object-oriented database management systems are in fact different things, but they share one key concept: that software or data can be "containerized." Everything goes into a box, and there can be boxes within boxes. In the simplest conventional program, one program step equates to one instruction; in an object-oriented language, each step might be a whole boxful of instructions. Similarly, with an object-oriented database, one variable, instead of equating to a single data element, may equate to a whole boxful of data.

Table D.1 introduces some of the key terms used in object-oriented design.

Table D.1 Key Object-Oriented Terms

Term	Definition
Attribute	Data variables contained within an object.
Containment	A relationship between two object instances in which the containing object includes a pointer to the contained object.
Encapsulation	The isolation of the attributes and services of an object instance from the external environment. Services may only be invoked by name and attributes may only be accessed by means of the services.
Inheritance	A relationship between two object classes in which the attributes and services of a parent class are acquired by a child class.
Interface	A description closely related to an object class. An interface contains method definitions (without implementations) and constant values. An interface cannot be instantiated as an object.
Message	The means by which objects interact.
Method	A procedure that is part of an object and that can be activated from outside the object to perform certain functions.
Object	An abstraction of a real-world entity.
Object class	A named set of objects that share the same names, sets of attributes, and services.
Object instance	A specific member of an object class, with values assigned to the attributes.
Polymorphism	Refers to the existence of multiple objects that use the same names for services and present the same interface to the external world but that represent different types of entities.
Service	A function that performs an operation on an object.

D.2 OBJECT-ORIENTED CONCEPTS

The central concept of object-oriented design is the object. An object is a distinct software unit that contains a collection of related variables (data) and methods (procedures). Generally, these variables and methods are not directly visible outside the object. Rather, well-defined interfaces exist that allow other software to have access to the data and the procedures.

An object represents some thing, be it a physical entity, a concept, a software module, or some dynamic entity such as a TCP connection. The values of the variables in the object express the information that is known about the thing that the object represents. The methods include procedures whose execution affect the values in the object and possibly also affect that thing being represented.

Figures D.1 and D.2 illustrate key object-oriented concepts.

Object Structure

The data and procedures contained in an object are generally referred to as variables and methods, respectively. Everything that an object "knows" can be expressed in its variables, and everything it can do is expressed in its methods.

The **variables** in an object, also called **attributes**, are typically simple scalars or tables. Each variable has a type, possibly a set of allowable values, and may either be constant or variable (by convention, the term *variable* is used even for constants).

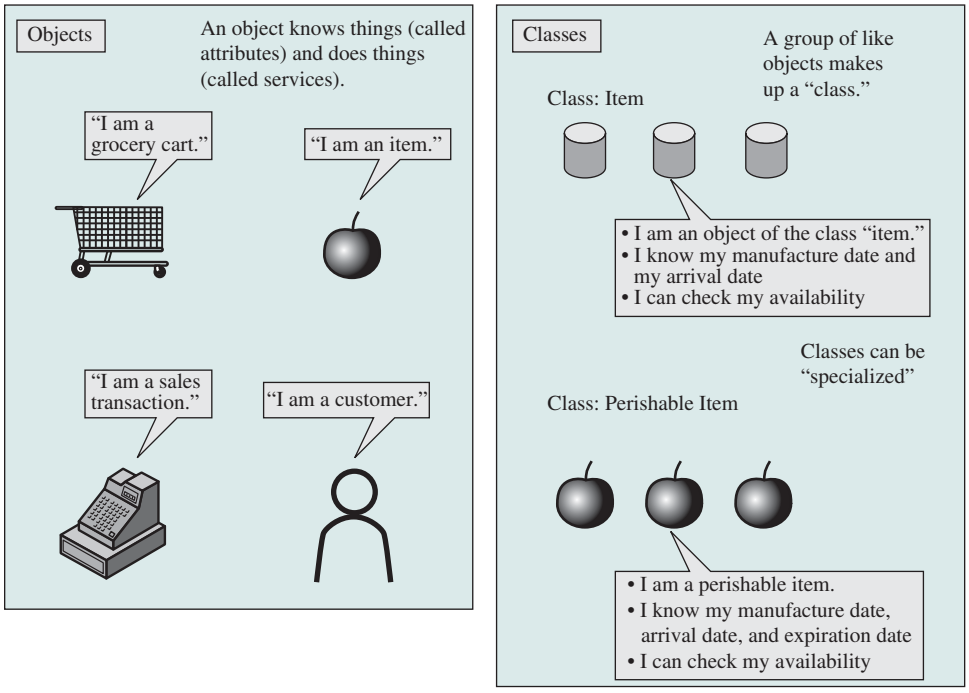


Figure D.1 Objects

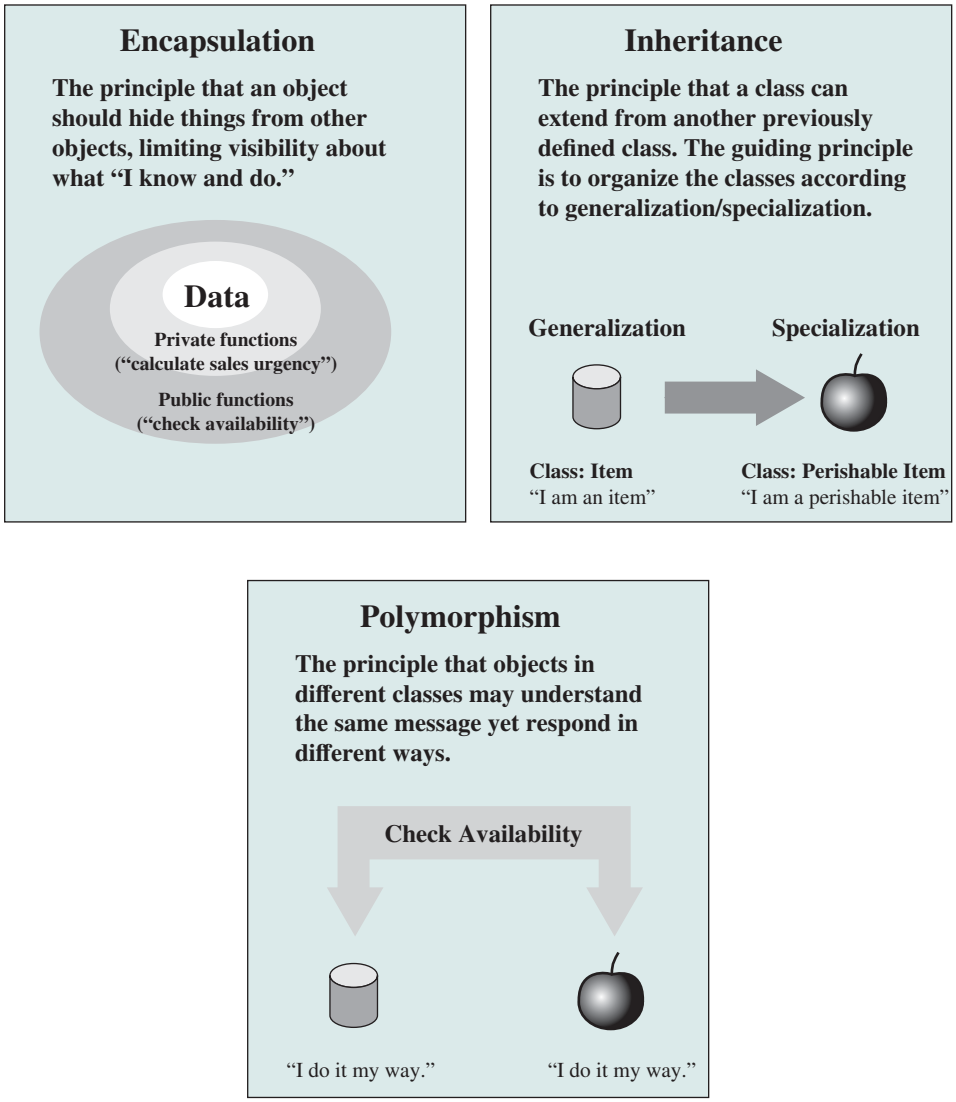


Figure D.2 Object Concepts

Access restrictions may also be imposed on variables for certain users, classes of users, or situations.

The **methods** in an object are procedures that can be triggered from outside to perform certain functions. The method may change the state of the object, update some of its variables, or act on outside resources to which the object has access.

Objects interact by means of **messages**. A message includes the name of the sending object, the name of the receiving object, the name of a method in the receiving object, and any parameters needed to qualify the execution of the method. A message can only be used to invoke a method within an object. The only

way to access the data inside an object is by means of the object's methods. Thus, a method may cause an action to be taken, or for the object's variables to be accessed, or both. For local objects, passing a message to an object is the same as calling an object's method. When objects are distributed, passing a message is exactly what it sounds like.

The interface of an object is a set of public methods that the object supports. An interface says nothing about implementation; objects in different classes may have different implementations of the same interfaces.

The property of an object that its only interface with the outside world is by means of messages is referred to as **encapsulation**. The methods and variables of an object are encapsulated and available only via message-based communication. Encapsulation offers two advantages:

1. It protects an object's variables from corruption by other objects. This protection may include protection from unauthorized access and protection from the types of problems that arise from concurrent access, such as deadlock and inconsistent values.
2. It hides the internal structure of the object so that interaction with the object is relatively simple and standardized. Furthermore, if the internal structure or procedures of an object are modified without changing its external functionality, other objects are unaffected.

Object Classes

In practice, there will typically be a number of objects representing the same types of things. For example, if a process is represented by an object, then there will be one object for each process present in a system. Clearly, every such object needs its own set of variables. However, if the methods in the object are reentrant procedures, then all similar objects could share the same methods. Furthermore, it would be inefficient to redefine both methods and variables for every new but similar object.

The solution to these difficulties is to make a distinction between an object class and an object instance. An **object class** is a template that defines the methods and variables to be included in a particular type of object. An **object instance** is an actual object that includes the characteristics of the class that defines it. The object contains values for the variables defined in the object class. **Instantiation** is the process of creating a new object instance for an object class.

INHERITANCE The concept of an object class is powerful because it allows for the creation of many object instances with a minimum of effort. This concept is made even more powerful by the use of the mechanism of inheritance [TAIV96].

Inheritance enables a new object class to be defined in terms of an existing class. The new (lower level) class, called the **subclass**, or the **child class**, automatically includes the methods and variable definitions in the original (higher-level) class, called the **superclass**, or **parent class**. A subclass may differ from its superclass in a number of ways:

1. The subclass may include additional methods and variables not found in its superclass.

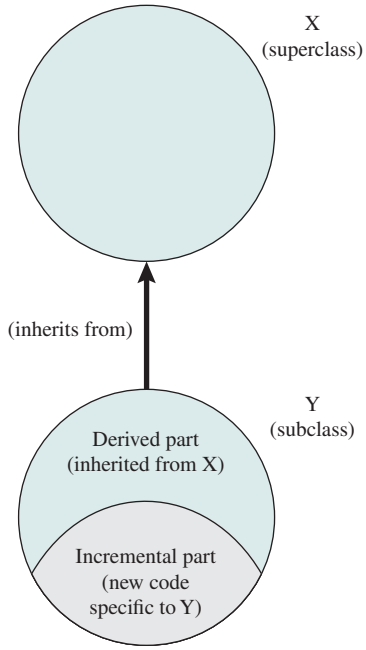


Figure D.3 Inheritance

2. The subclass may override the definition of any method or variable in its superclass by using the same name with a new definition. This provides a simple and efficient way of handling special cases.
3. The subclass may restrict a method or variable inherited from its superclass in some way.

Figure D.3, based on one in [KORS90], illustrates the concept.

The inheritance mechanism is recursive, allowing a subclass to become the superclass of its own subclasses. In this way, an **inheritance hierarchy** may be constructed. Conceptually, we can think of the inheritance hierarchy as defining a search technique for methods and variables. When an object receives a message to carry out a method that is not defined in its class, it automatically searches up the hierarchy until it finds the method. Similarly, if the execution of a method results in the reference to a variable not defined in that class, the object searches up the hierarchy for the variable name.

POLYMORPHISM Polymorphism is an intriguing and powerful characteristic that makes it possible to hide different implementations behind a common interface. Two objects that are polymorphic to each other utilize the same names for methods and present the same interface to other objects. For example, there may be a number of print objects for different output devices, such as `printDotmatrix`, `printLaser`, `printScreen`, and so forth, or for different types of documents, such as `printText`, `printDrawing`, and `printCompound`. If each such object includes a method called `print`, then any document could be printed by sending

the message print to the appropriate object, without concern for how that method is actually carried out. Typically, polymorphism is used to allow you have the same method in multiple subclasses of the same superclass, each with a different detailed implementation.

It is instructive to compare polymorphism to the usual modular programming techniques. An objective of top-down modular design is to design lower-level modules of general utility with a fixed interface to higher-level modules. This allows the one lower-level module to be invoked by many different higher-level modules. If the internals of the lower-level module are changed without changing its interface, then none of the upper-level modules that use it are affected. By contrast, with polymorphism, we are concerned with the ability of one higher-level object to invoke many different lower-level objects using the same message format to accomplish similar functions. With polymorphism, new lower-level objects can be added with minimal changes to existing objects.

INTERFACES Inheritance enables a subclass object to use functionality of a superclass. There may be cases when you wish to define a subclass that has functionality of more than one superclass. This could be accomplished by allowing a subclass to inherit from more than one superclass. C++ is one language that allows such multiple inheritance. However, for simplicity, most modern object-oriented languages including Java, C\#, and Visual Basic .NET limit a class to inheriting from only one superclass. Instead, a feature known as *interfaces* is used to enable a class to borrow some functionality from one class and other functionality from a completely different class.

Unfortunately, the term *interface* is used in much of the literature on objects with both a general-purpose and a specific functional meaning. An interface, as we are discussing it here, specifies an application-programming interface (API) for certain functionality. It does not define any implementation for that API. The syntax for an interface definition typically looks similar to a class definition, except that there is no code defined for the methods, just the method names, the arguments passed, and the type of the value returned. An interface may be implemented by a class. This works in much the same way that inheritance works. If a class implements an interface, it must have the properties and methods of the interface defined in the class. The methods that are implemented can be coded in any fashion, so long as the name, arguments, and return type of each method from the interface are identical to the definition in the interface.

Containment

Object instances that contain other objects are called **composite objects**. Containment may be achieved by including the pointer to one object as a value in another object. The advantage of composite objects is that they permit the representation of complex structures. For example, an object contained in a composite object may itself be a composite object.

Typically, the structures built up from composite objects are limited to a tree topology; that is, no circular references are allowed and each "child" object instance may have only one "parent" object instance.

It is important to be clear about the distinction between an inheritance hierarchy of object classes, and a containment hierarchy of object instances. The two are not related. The use of inheritance simply allows many different object types to be defined with a minimum of efforts. The use of containment allows the construction of complex data structures.

D.3 BENEFITS OF OBJECT-ORIENTED DESIGN

[CAST92] lists the following benefits of object-oriented design:

- **Better organization of inherent complexity:** Through the use of inheritance, related concepts, resources, and other objects can be efficiently defined. Through the use of containment, arbitrary data structures, which reflect the underlying task at hand, can be constructed. Object-oriented programming languages and data structures enable designers to describe operating system resources and functions in a way that reflects the designer's understanding of those resources and functions.
- **Reduced development effort through reuse:** Reusing object classes that have been written, tested, and maintained by others reduces development, testing, and maintenance time.
- **More extensible and maintainable systems:** Maintenance, including product enhancements and repairs, traditionally consumes about 65% of the cost of any product life cycle. Object-oriented design drives that percentage down. The use of object-based software helps limit the number of potential interactions of different parts of the software, ensuring changes to the implementation of a class can be made with little impact on the rest of the system.

These benefits are driving operating system design in the direction of object-oriented systems. Objects enable programmers to customize an operating system to meet new requirements without disrupting system integrity. Objects also pave the road to distributed computing. Because objects communicate by means of messages, it matters not whether two communicating objects are on the same system or on two different systems in a network. Data, functions, and threads can be dynamically assigned to workstations and servers as needed. Accordingly, the object-oriented approach to the design of operating systems is becoming increasingly evident in PC and workstation operating systems.

D.4 CORBA

As we have seen in this book, object-oriented concepts have been used to design and implement operating system kernels, bringing benefits of flexibility, manageability, and portability. The benefits of using object-oriented techniques extend with equal or greater benefit to the realm of distributed software, including distributed operating systems. The application of object-oriented techniques to the design and implementation of distributed software is referred to as distributed object computing (DOC).

The motivation for DOC is the increasing difficulty in writing distributed software: while computing and network hardware get smaller, faster, and cheaper, distributed software gets larger, slower, and more expensive to develop and maintain. [SCHM97] points out that the challenge of distributed software stems from two types of complexity:

- **Inherent:** Inherent complexities arise from fundamental problems of distribution. Chief among these are detecting and recovering from network and host failures, minimizing the impact of communication latency, and determining an optimal partitioning of service components and workload onto computers throughout a network. In addition, concurrent programming, with issues of resource locking and deadlocks, is still difficult, and distributed systems are inherently concurrent.
- **Accidental:** Accidental complexities arise from limitations with tools and techniques used to build distributed software. A common source of accidental complexity is the widespread use of functional design, which results in nonextensible and nonreusable systems.

DOC is a promising approach to managing both types of complexity. The centerpiece of the DOC approach are object request brokers (ORBs), which act as intermediaries for communication between local and remote objects. ORBs eliminate some of the tedious, error-prone, and nonportable aspects of designing and implementing distributed applications. Supplementing the ORB must be a number of conventions and formats for message exchange and interface definition between applications and the object-oriented infrastructure.

There are three main competing technologies in the DOC market: the object management group (OMG) architecture, called Common Object Request Broker Architecture (CORBA); the Java remote method invocation (RMI) system; and Microsoft's distributed component object model (DCOM). CORBA is the most advanced and well-established of the three. A number of industry leaders, including IBM, Sun, Netscape, and Oracle, support CORBA, and Microsoft has announced that it will link its Windows-only DCOM with CORBA. The remainder of this appendix provides a brief overview of CORBA.

Table D.2 defines some key terms used in CORBA. The main features of CORBA are as follows (see Figure D.4):

- **Clients:** Clients generate requests and access object services through a variety of mechanisms provided by the underlying ORB.
- **Object implementations:** These implementations provide the services requested by various clients in the distributed system. One benefit of the CORBA architecture is that both clients and object implementations can be written in any number of programming languages and can still provide the full range of required services.
- **ORB core:** The ORB core is responsible for communication between objects. The ORB finds an object on the network, delivers requests to the object, activates the object (if not already active), and returns any message back to the sender. The ORB core provides **access transparency** because programmers use

Table D.2 Key Concepts in a Distributed CORBA System

CORBA Concept	Definition
Client application	Invokes requests for a server to perform operations on objects. A client application uses one or more interface definitions that describe the objects and operations the client can request. A client application uses object references, not objects, to make requests.
Exception	Contains information that indicates whether a request was successfully performed.
Implementation	Defines and contains one or more methods that do the work associated with an object operation. A server can have one or more implementations.
Interface	Describes how instances of an object will behave, such as what operations are valid on those objects.
Interface definition	Describes the operations that are available on a certain type of object.
Invocation	The process of sending a request.
Method	The server code that does the work associated with an operation. Methods are contained within implementations.
Object	Represents a person, place, thing, or piece of software. An object can have operations performed on it, such as the promote operation on an employee object.
Object instance	An occurrence of one particular kind of object.
Object reference	An identifier of an object instance.
OMG Interface Definition Language (IDL)	A definition language for defining interfaces in CORBA.
Operation	The action that a client can request a server to perform on an object instance.
Request	A message sent between a client and a server application.
Server application	Contains one or more implementations of objects and their operations.

exactly the same method with the same parameters when invoking a local method or a remote method. The ORB core also provides **location transparency**: Programmers do not need to specify the location of an object.

- **Interface:** An object's interface specifies the operations and types supported by the object, and thus defines the requests that can be made on the object. CORBA interfaces are similar to classes in C++ and interfaces in Java. Unlike C++ classes, a CORBA interface specifies methods and their parameters and return values, but is silent about their implementation. Two objects of the same C++ class have the same implementation of their methods.
- **OMG interface definition language (IDL):** IDL is the language used to define objects. An example IDL interface definition is:

```
//OMG IDL
interface Factory
    { Object create ( ) ;
} ;
```

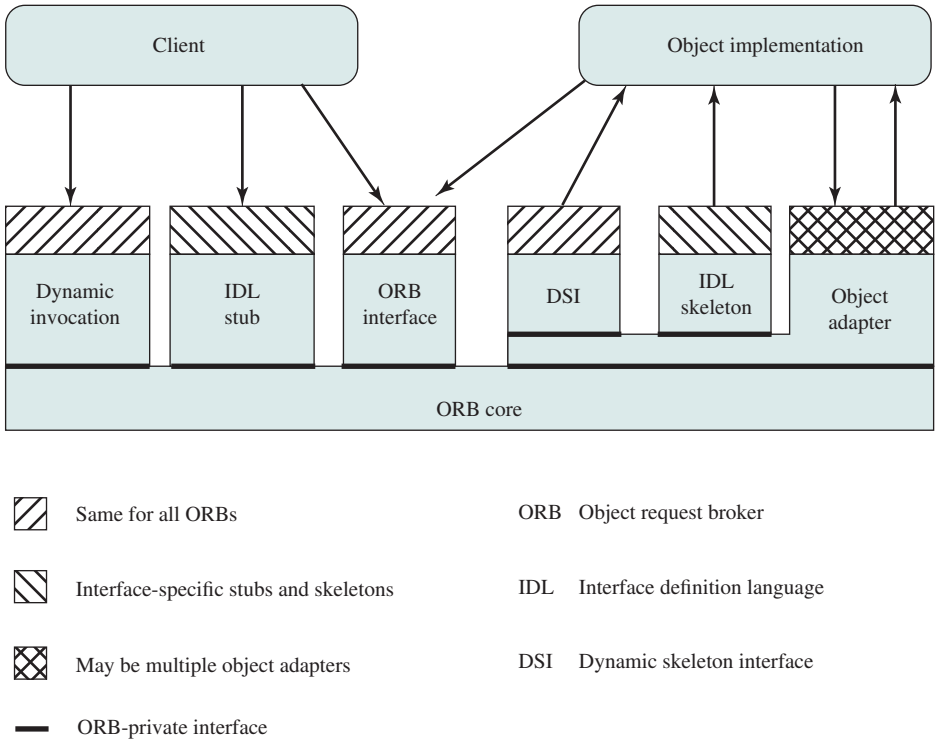


Figure D.4 Common Object Request Broker Architecture

This definition specifies an interface named `Factory` that supports one operation, `create`. The create operation takes no parameters and returns an object reference of type `Object`. Given an object reference for an object of type `Factory`, a client could invoke it to create a new CORBA object. IDL is a programming-independent language and, for this reason, a client does not invoke directly any object operation. It needs a mapping to the client programming language to do that. It is also possible, that the server and the client are programmed in different programming languages. The use of a specification language is a way to deal with heterogeneous processing across multiple languages and platform environments. Thus, IDL enables **platform independence**.

- **Language binding creation:** IDL compilers map one OMG IDL file to different programming languages, which may or may not be object oriented, such as Java, Smalltalk, Ada, C, C++, and COBOL. That mapping includes the definition of the language-specific data types and procedure interfaces to access service objects, the IDL client stub interface, the IDL skeleton, the object adapters, the dynamic skeleton interface, and the direct ORB interface. Usually, clients have a compile-time knowledge of the object interface and use client stubs to do a static invocation; in certain cases, clients do not have that knowledge and they must do a dynamic invocation.

- **IDL stub:** Makes calls to the ORB core on behalf of a client application. IDL stubs provide a set of mechanisms that abstract the ORB core functions into direct RPC (remote procedure call) mechanisms that can be employed by the end-client applications. These stubs make the combination of the ORB and remote object implementation appear as if they were tied to the same in-line process. In most cases, IDL compilers generate language-specific interface libraries that complete the interface between the client and object implementations.
- **IDL skeleton:** Provides the code that invokes specific server methods. Static IDL skeletons are the server-side complements to the client-side IDL stubs. They include the bindings between the ORB core and the object implementations that complete the connection between the client and object implementations.
- **Dynamic invocation:** Using the dynamic invocation interface (DII), a client application can invoke requests on any object without having compile-time knowledge of the object's interfaces. The interface details are filled in by consulting with an interface repository and/or other run-time sources. The DII allows a client to issue one-way commands (for which there is no response).
- **Dynamic skeleton interface (DSI):** Similar to the relationship between IDL stubs and static IDL skeletons, the DSI provides dynamic dispatch to objects. Equivalent to dynamic invocation on the server side.
- **Object adapter:** An object adapter is CORBA system component provided by the CORBA vendor to handle general ORB-related tasks, such as activating objects and activating implementations. The adapter takes these general tasks and ties them to particular implementations and methods in the server.

D.5 RECOMMENDED READING AND WEBSITE

[KORS90] is a good overview of object-oriented concepts. [STRO88] is a clear description of object-oriented programming. An interesting perspective on object-oriented concepts is provided in [SYND93]. [VINO97] is an overview of CORBA.

KORS90 Korson, T., and McGregor, J. "Understanding Object-Oriented: A Unifying Paradigm." *Communications of the ACM*, September 1990.

STRO88 Stroustrup, B. "What is Object-Oriented Programming?" *IEEE Software*, May 1988.

SNYD93 Snyder, A. "The Essence of Objects: Concepts and Terms." *IEEE Software*, January 1993.

VINO97 Vinoski, S. "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments." *IEEE Communications Magazine*, February 1997.

Recommended Website:

Object Management Group: Industry consortium that promotes CORBA and related object technologies

APPENDIX E

AMDAHL'S LAW

E.1 IMPLICATIONS OF AMDAHL'S LAW

When considering system performance, computer system designers look for ways to improve performance by improvement in technology or change in design. Examples include the use of parallel processors, the use of a memory cache hierarchy, and speedup in memory access time and I/O transfer rate due to technology improvements. In all of these cases, it is important to note that a speedup in one aspect of the technology or design does not result in a corresponding improvement in performance. This limitation is succinctly expressed by Amdahl's law.

Amdahl's law was first proposed by Gene Amdahl in 1967 [AMDA67] and deals with the potential speedup of a program using multiple processors compared to a single processor. Consider a program running on a single processor such that a fraction $(1 - f)$ of the execution time involves code that is inherently serial, and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead. Let T be the total execution time of the program using a single processor. Then the speedup using a parallel processor with N processors that fully exploits the parallel portion of the program is as follows:

$$\text{Speedup} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}}$$

$$= \frac{T(1 - f) + Tf}{T(1 - f) + \frac{Tf}{N}} = \frac{1}{(1 - f) + \frac{f}{N}}$$

This equation is illustrated in Figure E.1. Two important conclusions can be drawn:

1. When f is small, the use of parallel processors has little effect.
2. As N approaches infinity, speedup is bound by $1/(1 - f)$, so there are diminishing returns for using more processors.

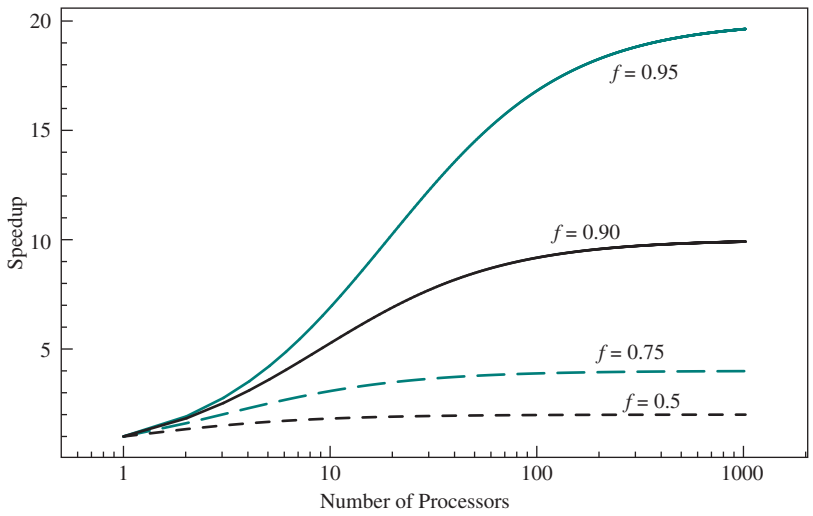


Figure E.1 Amdahl's Law for Multiprocessors

These conclusions are too pessimistic, an assertion first put forward in [GUST88]. For example, a server can maintain multiple threads or multiple tasks to handle multiple clients and execute the threads or tasks in parallel up to the limit of the number of processors. Many database applications involve computations on massive amounts of data that can be split up into multiple parallel tasks. Nevertheless, Amdahl's law illustrates the problems facing industry in the development of multicore machines with an ever-growing number of cores: The software that runs on such machines must be adapted to a highly parallel execution environment to exploit the power of parallel processing.

Amdahl's law can be generalized to evaluate any design or technical improvement in a computer system. Consider any enhancement to a feature of a system that results in a speedup. The speedup can be expressed as follows:

$$\text{Speedup} = \frac{\text{Performance after enhancement}}{\text{Performance before enhancement}} = \frac{\text{Execution time before enhancement}}{\text{Execution time after enhancement}}$$

Suppose a feature of the system is used during execution a fraction of the time f, before enhancement, and the speedup of that feature after enhancement is SU_f. Then the overall speedup of the system is

$$Speedup = \frac{1}{(1 - f) + \frac{f}{SU_f}}$$

For example, suppose a task makes extensive use of floating-point operations, with 40% of the time is consumed by floating-point operations. With a new hardware design, the floating-point module is speeded up by a factor of K. Then, the overall speedup is:

$$Speedup = \frac{1}{0.6 + \frac{0.4}{K}}$$

Thus, independent of K, the maximum speedup is 1.67.

E.2 REFERENCES

AMDA67 Amdahl, G. "Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capability." *Proceedings, of the AFIPS Conference*, 1967.

GUST88 Gustafson, J. "Reevaluating Amdahl's Law." *Communications of the ACM*, May 1988.

APPENDIX F

HASH TABLES

Consider the following problem. A set of N items is to be stored in a table. Each item consists of a label plus some additional information, which we can refer to as the value of the item. We would like to be able to perform a number of ordinary operations on the table, such as insertion, deletion, and searching for a given item by label.

If the labels of the items are numeric, in the range 0 to $M - 1$, then a simple solution would be to use a table of length M. An item with label i would be inserted into the table at location i. As long as items are of fixed length, table lookup is trivial and involves indexing into the table based on the numeric label of the item. Furthermore, it is not necessary to store the label of an item in the table, because this is implied by the position of the item. Such a table is known as a **direct access table**.

If the labels are nonnumeric, then it is still possible to use a direct access approach. Let us refer to the items as A[1], . . . A[N]. Each item A[i] consists of a label, or key, k_i, and a value v_i. Let us define a mapping function I(k) such that I(k) takes a value between 1 and M for all keys, and I(k_i) $\neq$ I(k_j) for any i and j. In this case, a direct access table can also be used, with the length of the table equal to M.

The one difficulty with these schemes occurs if M is much greater than N. In this case, the proportion of unused entries in the table is large, and this is an inefficient use of memory. An alternative would be to use a table of length N and store the N items (label plus value) in the N table entries. In this scheme, the amount of memory is minimized, but there is now a processing burden to do table lookup. There are several possibilities:

- **Sequential search**: This brute-force approach is time consuming for large tables.
- **Associative search**: With the proper hardware, all of the elements in a table can be searched simultaneously. This approach is not general purpose and cannot be applied to any and all tables of interest.
- **Binary search**: If the labels or the numeric mapping of the labels are arranged in ascending order in the table, then a binary search is much quicker than sequential (see Table F.1) and requires no special hardware.

The binary search looks promising for table lookup. The major drawback with this method is that adding new items is not usually a simple process and will require reordering of the entries. Therefore, binary search is usually used only for reasonably static tables that are seldom changed.

Table F.1 Average Search Length for One of N items in a Table of Length M

Technique	Search Length
Direct	1
Sequential	$\frac{M + 1}{2}$
Binary	$\log_2 M$
Linear hashing	$\frac{2 - N/M}{2 - 2N/M}$
Hash (overflow with chaining)	$1 + \frac{N - 1}{2M}$

We would like to avoid the memory penalties of a simple direct access approach and the processing penalties of the alternatives listed previously. The most frequently used method to achieve this compromise is **hashing**. Hashing, which was developed in the 1950s, is simple to implement and has two advantages. First, it can find most items with a single seek, as in direct accessing. Second, insertions and deletions can be handled without added complexity.

The hashing function can be defined as follows. Assume up to N items are to be stored in a **hash table** of length M, with $M \geq N$, but not much larger than N. To insert an item in the table:

I1. Convert the label of the item to a near-random number n between 0 and $M - 1$. For example, if the label is numeric, a popular mapping function is to divide the label by M and take the remainder as the value of n.

I2. Use n as the index into the hash table.

- **a.** If the corresponding entry in the table is empty, store the item (label and value) in that entry.
- **b.** If the entry is already occupied, then store the item in an overflow area, as discussed subsequently.

To perform table lookup of an item whose label is known:

L1. Convert the label of the item to a near-random number n between 0 and $M - 1$, using the same mapping function as for insertion.

L2. Use n as the index into the hash table.

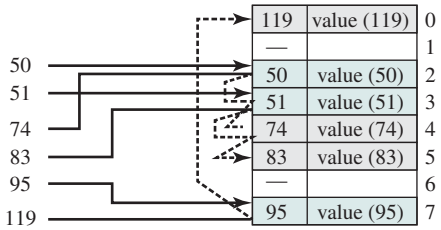
- **a.** If the corresponding entry in the table is empty, then the item has not previously been stored in the table.
- **b.** If the entry is already occupied and the labels match, then the value can be retrieved.
- **c.** If the entry is already occupied and the labels do not match, then continue the search in the overflow area.

Hashing schemes differ in the way in which the overflow is handled. One common technique is referred to as the **linear hashing** technique and is commonly used in compilers. In this approach, rule I2.b becomes

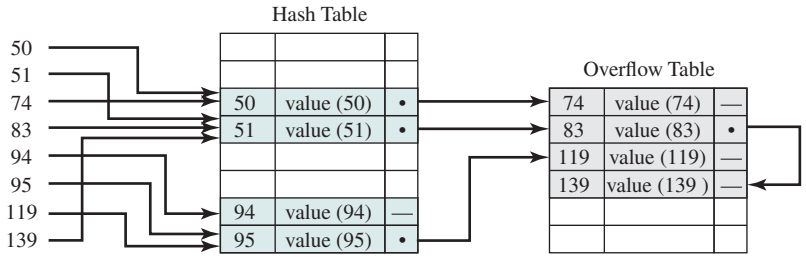
I2.b. If the entry is already occupied, set $n = n + 1 \pmod{M}$ and go back to step I2.a.

Rule L2.c is modified accordingly.

Figure F.1a is an example. In this case, the labels of the items to be stored are numeric, and the hash table has eight positions ($M = 8$). The mapping function is to take the remainder upon division by 8. The figure assumes the items were inserted in ascending numerical order, although this is not necessary. Thus, items 50 and 51 map into positions 2 and 3, respectively, and as these are empty, they are inserted there. Item 74 also maps into position 2, but as it is not empty, position 3 is tried. This is also occupied, so the position 4 is ultimately used.



(a) Linear rehashing



(b) Overflow with chaining

Figure F.1 Hashing

It is not easy to determine the average length of the search for an item in an open hash table because of the clustering effect. An approximate formula was obtained by Schay and Spruth:[1]

$$\text{Average search length} = \frac{2 - r}{2 - 2r}$$

where $r = N/M$. Note the result is independent of table size and depends only on how full the table is. The surprising result is with the table 80% full, the average length of the search is still around 3.

Even so, a search length of 3 may be considered long, and the linear hashing table has the additional problem that it is not easy to delete items. A more attractive approach, which provides shorter search lengths (see Table F.1) and allows deletions as well as additions, is **overflow with chaining**. This technique is illustrated in Figure F.1b. In this case, there is a separate table into which overflow entries are inserted. This table includes pointers passing down the chain of entries associated with any position in the hash table. In this case, the average search length, assuming randomly distributed data, is

$$\text{Average search length} = 1 + \frac{N - 1}{2M}$$

For large values of N and M, this value approaches 1.5 for $N = M$. Thus, this technique provides for compact storage with rapid lookup.

[1]Schay, G., and Spruth, W. "Analysis of a File Addressing Method." *Communications of the ACM,* August 1962.

APPENDIX G

RESPONSE TIME

G.1 RESPONSE TIME CONSIDERATIONS

Response time is the time taken by a system to react to a given input. In an interactive transaction, it may be defined as the time between the last keystroke by the user and the beginning of the display of a result by the computer. For different types of applications, a slightly different definition is needed. In general, it is the time taken for the system to respond to a request to perform a particular task.

Ideally, one would like the response time for any application to be short. However, it is almost invariably the case that shorter response time imposes greater cost. This cost comes from two sources:

- **Computer processing power:** The faster the processor is, the shorter the response time will be. Of course, increased processing power means increased cost.
- **Competing requirements:** Providing rapid response time to some processes may penalize other processes.

Thus the value of a given level of response time must be assessed versus the cost of achieving that response time.

Table G.1, from [MART88] lists six general ranges of response times. Design difficulties are faced when a response time of less than 1 second is required. A requirement for a sub-second response time is generated by a system that controls or in some other way interacts with an ongoing external activity, such as an assembly line. Here the requirement is straightforward. When we consider human-computer interaction, such as in a data entry application, then we are in the realm of conversational response time. In this case, there is still a requirement for a short response time, but the acceptable length of time may be difficult to assess.

That rapid response time is the key to productivity in interactive applications has been confirmed in a number of studies [SHNE84; THAD81; GUYN88]. These studies show that when a computer and a user interact at a pace that ensures neither has to wait on the other, productivity increases significantly, the cost of the work done on the computer therefore drops, and quality tends to improve. It used to be widely accepted that a relatively slow response, up to 2 seconds, was acceptable for most interactive applications because the person was thinking about the next task. However, it now appears that productivity increases as rapid response times are achieved.

The results reported on response time are based on an analysis of online transactions. A transaction consists of a user command from a terminal and the system's reply. It is the fundamental unit of work for online system users. It can be divided into two time sequences:

- **User response time:** The time span between the moment the user receives a complete reply to one command and enters the next command. People often refer to this as think time.
- **System response time:** The time span between the moment the user enters a command and the moment a complete response is displayed on the terminal.

Table G.1 Response Time Ranges

Greater than 15 seconds
This rules out a conversational interaction. For certain types of applications, certain types of users may be content to sit at a terminal for more than 15 seconds waiting for the answer to a single simple inquiry. However, for a busy person, captivity for more than 15 seconds seems intolerable. If such delays will occur, the system should be designed so the user can turn to other activities and request the response at some later time.
Greater than 4 seconds
These are generally too long for a conversation requiring the operator to retain information in short-term memory (the operator's memory, not the computer's!). Such delays would be very inhibiting in problem-solving activity and frustrating in data entry activity. However, after a major closure, such as the end of a transaction, delays from 4 to 15 seconds can be tolerated.
2 to 4 seconds
A delay longer than 2 seconds can be inhibiting to terminal operations demanding a high level of concentration. A wait of 2–4 seconds at a terminal can seem surprisingly long when the user is absorbed and emotionally committed to complete what he or she is doing. Again, a delay in this range may be acceptable after a minor closure has occurred.
Less than 2 seconds
When the terminal user has to remember information throughout several responses, the response time must be short. The more detailed the information remembered, the greater the need for responses of less than 2 seconds. For elaborate terminal activities, 2 seconds represents an important response-time limit.
Sub-second response time
Certain types of thought-intensive work, especially with graphics applications, require very short response times to maintain the user's interest and attention for long periods of time.
Decisecond response time
A response to pressing a key and seeing the character displayed on the screen or clicking a screen object with a mouse needs to be almost instantaneous—less than 0.1 second after the action. Interaction with a mouse requires extremely fast interaction if the designer is to avoid the use of alien syntax (one with commands, mnemonics, punctuation, etc.).

As an example of the effect of reduced system response time, Figure G.1 shows the results of a study carried out on engineers using a computer-aided design graphics program for the design of integrated circuit chips and boards [SMIT83]. Each transaction consists of a command by the engineer that in some way alters the graphic image being displayed on the screen. The results show that the rate of transactions increases as system response time falls and rises dramatically once system response time falls below 1 second. What is happening is that as the system response time falls, so does the user response time. This has to do with the effects of short-term memory and human attention span.

Another area where response time has become critical is the use of the World Wide Web, either over the Internet or over a corporate intranet. The time taken for a typical Web page to come up on the user's screen varies greatly. Response times can be gauged based on the level of user involvement in the session; in particular, systems with vary fast response times tend to command more user attention. In a study by Sevcik [SEVC96, SEVC02], illustrated in Figure G.2, Web systems with a 3-second or

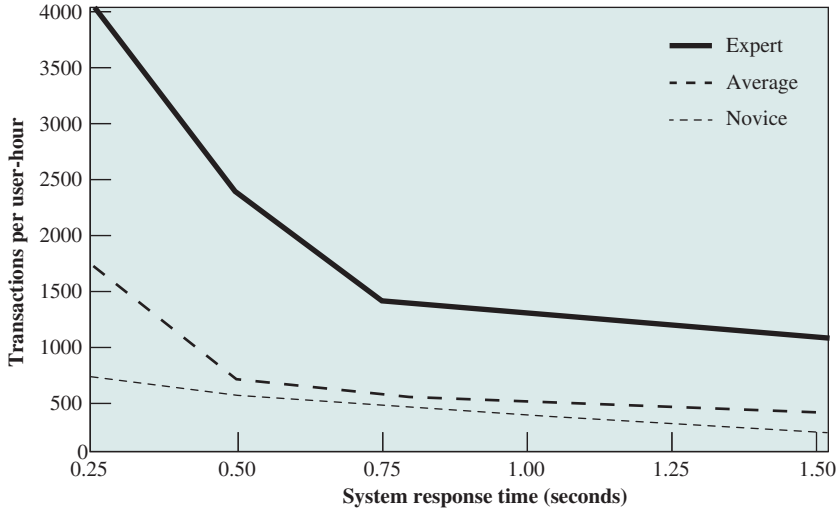


Figure G.1 Response Time Results for High-Function Graphics

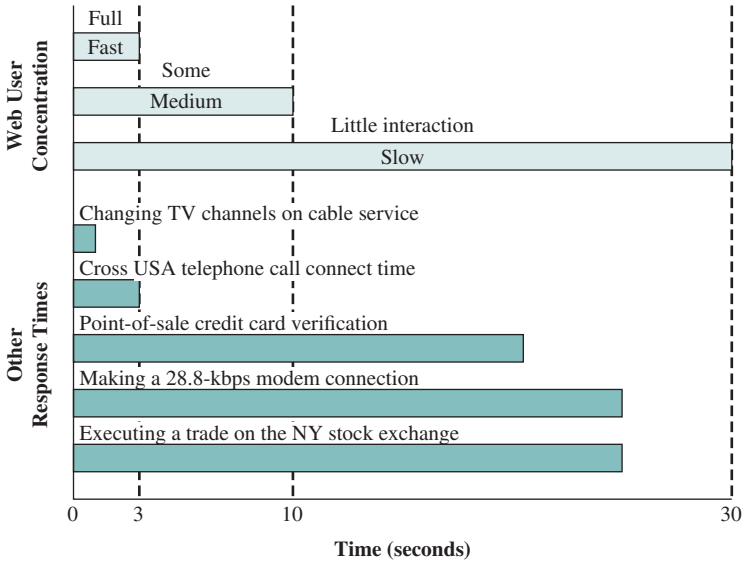


Figure G.2 Response Time Requirements

better response time maintain a high level of user attention. With a response time of between 3 and 10 seconds, some user concentration is lost, and response times above 10 seconds discourage the user, who may simply abort the session. Other studies of Web response time generally confirm these findings [BHAT01].

G.2 REFERENCES

BHAT01 Bhatti, N.; Bouch, A.; and Kuchinsky, A. "Integrated User-Perceived Quality into Web Server Design." *Proceedings, 9th International World Wide Web Conference*, May 2000.

GUYN88 Guynes, J. "Impact of System Response Time on State Anxiety." *Communications of the ACM,* March 1988.

MART88 Martin, J. *Principles of Data Communication.* Englewood Cliffs, NJ: Prentice Hall, 1988.

SELV99 Selvidge, P. "How Long Is Too Long to Wait for a Webpage to Load." *Usability News*, Wichita State University, July 1999.

SEVC96 Sevcik, P. "Designing a High-Performance Web Site." *Business Communications Review*, March 1996.

SEVC02 Sevcik, P. "Understanding How Users View Application Performance." *Business Communications Review*, July 2002.

SHNE84 Shneiderman, B. "Response Time and Display Rate in Human Performance with Computers." *ACM Computing Surveys*, September 1984.

SMIT83 Smith, D. "Faster Is Better: A Business Case for Subsecond Response Time." *Computerworld*, April 18, 1983.

THAD81 Thadhani, A. "Interactive User Productivity." *IBM Systems Journal*, No. 1, 1981.

APPENDIX H

QUEUEING SYSTEM CONCEPTS

In a number of chapters in this book, results from queueing theory are used. Chapter 21 provides a detailed discussion of queueing analysis. For purposes of understanding the description of the results in the book, however, the brief overview in this appendix should suffice. In this appendix, we present a brief definition of queueing systems and define key terms.

H.1 WHY QUEUEING ANALYSIS?

It is often necessary to make projections of performance on the basis of existing load information or on the basis of estimated load for a new environment. A number of approaches are possible:

1. Do an after-the-fact analysis based on actual values.
2. Make a simple projection by scaling up from existing experience to the expected future environment.
3. Develop an analytic model based on queueing theory.
4. Program and run a simulation model.

Option 1 is no option at all: we will wait and see what happens. This leads to unhappy users and to unwise purchases. Option 2 sounds more promising. The analyst may take the position that it is impossible to project future demand with any degree of certainty. Therefore, it is pointless to attempt some exact modeling procedure. Rather, a rough-and-ready projection will provide ballpark estimates. The problem with this approach is that the behavior of most systems under a changing load is not what one would intuitively expect. If there is an environment in which there is a shared facility (e.g., a network, a transmission line, and a time-sharing system), then the performance of that system typically responds in an exponential way to increases in demand.

Figure H.1 is a representative example. The upper line shows what typically happens to user response time on a shared facility as the load on that facility increases. The load is expressed as a fraction of capacity. Thus, if we are dealing with a router that is capable of processing and forwarding 1000 packets per second, then a load of 0.5 represents an arrival rate of 500 packets per second, and the response time is the amount of time it takes to retransmit any incoming packet. The lower line is a simple projection[1] based on knowledge of the behavior of the system up to a load of 0.5. Note while things appear rosy when the simple projection is made, performance on the system will in fact collapse beyond a load of about 0.8 to 0.9.

Thus, a more exact prediction tool is needed. Option 3 is to make use of an analytic model, which is one that can be expressed as a set of equations that can be solved to yield the desired parameters (response time, throughput, etc.). For computer, operating system, and networking problems, and indeed for many practical real-world problems, analytic models based on queueing theory provide a reasonably good fit to reality. The disadvantage of queueing theory is that a number of simplifying assumptions must be made to derive equations for the parameters of interest.

[1]The lower line is based on fitting a third-order polynomial to the data available up to a load of 0.5.

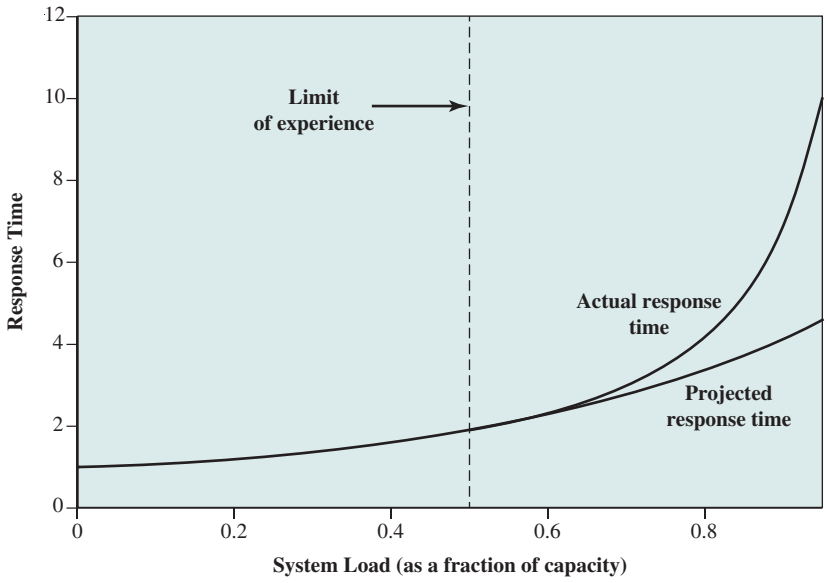


Figure H.1 **Projected versus Actual Response Time**

The final approach is a simulation model. Here, given a sufficiently powerful and flexible simulation programming language, the analyst can model reality in great detail and avoid making many of the assumptions required of queueing theory. However, in most cases, a simulation model is not needed or at least is not advisable as a first step in the analysis. For one thing, both existing measurements and projections of future load carry with them a certain margin of error. Thus, no matter how good the simulation model, the value of the results is limited by the quality of the input. For another, despite the many assumptions required of queueing theory, the results that are produced often come quite close to those that would be produced by a more careful simulation analysis. Furthermore, a queueing analysis can literally be accomplished in a matter of minutes for a well-defined problem, whereas simulation exercises can take days, weeks, or longer to program and run.

Accordingly, it behooves the analyst to master the basics of queueing theory.

H.2 THE SINGLE-SERVER QUEUE

The simplest queueing system is depicted in Figure H.2. The central element of the system is a server, which provides some service to items. Items from some population of items arrive at the system to be served. If the server is idle, an item is served immediately. Otherwise, an arriving item joins a waiting line.[2] When the server has completed serving an item, the item departs. If there are items waiting in the queue, one

[2]The waiting line is referred to as a queue in some treatments in the literature; it is also common to refer to the entire system as a queue. Unless otherwise noted, we use the term *queue* to mean waiting line.

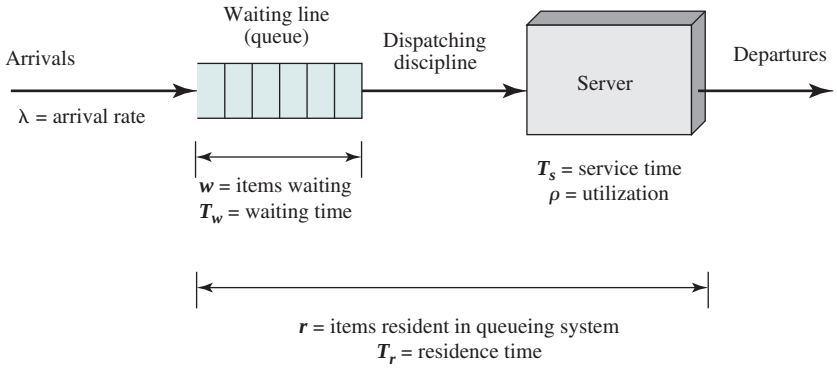


Figure H.2 Queueing System Structure and Parameters for Single-Server Queue

Table H.1 Notation for Queueing Systems

λ = arrival rate; mean number of arrivals per second
T_s = mean service time for each arrival; amount of time being served, not counting time waiting in the queue
ρ = utilization; fraction of time facility (server or servers) is busy
w = mean number of items waiting to be served
T_w = mean waiting time (including items that have to wait and items with waiting time = 0)
r = mean number of items resident in system (waiting and being served)
T_r = mean residence time; time an item spends in system (waiting and being served)

is immediately dispatched to the server. The server in this model can represent anything that performs some function or service for a collection of items. Some examples are: a processor provides service to processes; a transmission line provides a transmission service to packets or frames of data; an I/O device provides a read or write service for I/O requests.

Table H.1 summarizes some important parameters associated with a queueing model. Items arrive at the facility at some average rate (items arriving per second) λ. At any given time, a certain number of items will be waiting in the queue (zero or more); the average number waiting is w, and the mean time that an item must wait is T_w. T_w is averaged over all incoming items, including those that do not wait at all. The server handles incoming items with an average service time T_s; this is the time interval between the dispatching of an item to the server, and the departure of that item from the server. Utilization, ρ, is the fraction of time that the server is busy, measured over some interval of time. Finally, two parameters apply to the system as a whole. The average number of items resident in the system, including the item being served (if any) and the items waiting (if any), is r; and the average time that an item spends in the system, waiting and being served, is T_r; we refer to this as the mean residence time.[3]

[3]Again, in some of the literature, this is referred to as the mean queueing time, while other treatments use mean queueing time to mean the average time spent waiting in the queue (before being served).

If we assume that the capacity of the queue is infinite, then no items are ever lost from the system; they are just delayed until they can be served. Under these circumstances, the departure rate equals the arrival rate. As the arrival rate increases, the utilization increases and with it, congestion. The queue becomes longer, increasing waiting time. At $\rho = 1$, the server becomes saturated, working 100% of the time. Thus, the theoretical maximum input rate that can be handled by the system is

$$\lambda_{\max} = \frac{1}{T_s}$$

However, queues become very large near system saturation, growing without bound when $\rho = 1$. Practical considerations, such as response time requirements or buffer sizes, usually limit the input rate for a single server to between 70 and 90% of the theoretical maximum.

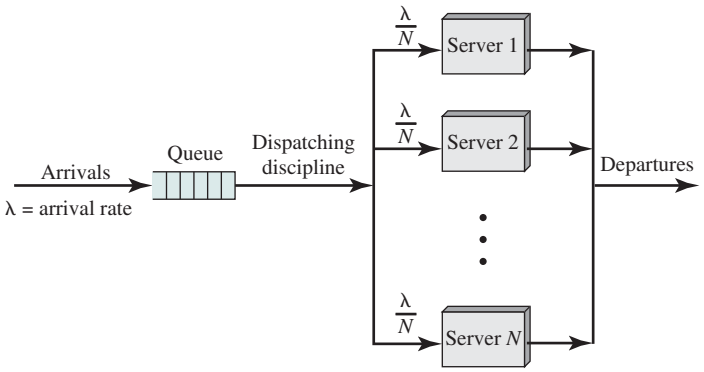
The following assumptions are typically made:

- **Item population:** Typically, we assume an infinite population. This means the arrival rate is not altered by the loss of population. If the population is finite, then the population available for arrival is reduced by the number of items currently in the system; this would typically reduce the arrival rate proportionally.
- **Queue size:** Typically, we assume an infinite queue size. Thus, the waiting line can grow without bound. With a finite queue, it is possible for items to be lost from the system. In practice, any queue is finite. In many cases, this will make no substantive difference to the analysis.
- **Dispatching discipline:** When the server becomes free, and if there is more than one item waiting, a decision must be made as to which item to dispatch next. The simplest approach is first-in-first-out; this discipline is what is normally implied when the term *queue* is used. Another possibility is last-in-first-out. One that you might encounter in practice is a dispatching discipline based on service time. For example, a packet-switching node may choose to dispatch packets on the basis of shortest first (to generate the most outgoing packets) or longest first (to minimize processing time relative to transmission time). Unfortunately, a discipline based on service time is very difficult to model analytically.

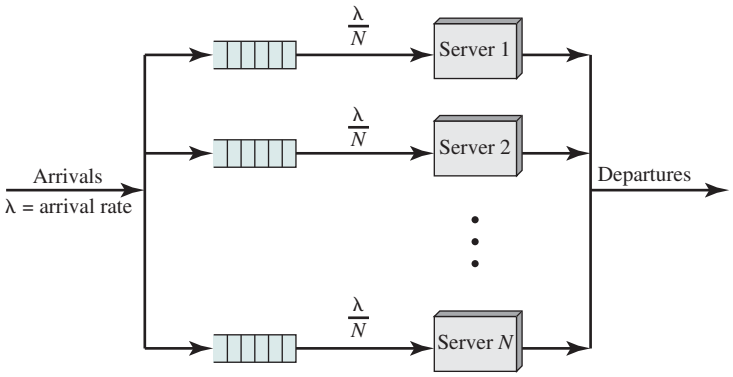
H.3 THE MULTISERVER QUEUE

Figure H.3 shows a generalization of the simple model we have been discussing for multiple servers, all sharing a common queue. If an item arrives and at least one server is available, then the item is immediately dispatched to that server. It is assumed all servers are identical; thus, if more than one server is available, it makes no difference which server is chosen for the item. If all servers are busy, a queue begins to form. As soon as one server becomes free, an item is dispatched from the queue using the dispatching discipline in force.

With the exception of utilization, all of the parameters illustrated in Figure H.2 carry over to the multiserver case with the same interpretation. If we have N identical



(a) Multiserver queue



(b) Multiple single-server queues

Figure H.3 Multiserver Versus Multiple Single-Server Queues

servers, then ρ is the utilization of each server, and we can consider N_ρ to be the utilization of the entire system; this latter term is often referred to as the traffic intensity, u. Thus, the theoretical maximum utilization is $N \times 100\%$, and the theoretical maximum input rate is:

$$\lambda_{\max} = \frac{N}{T_s}$$

The key characteristics typically chosen for the multiserver queue correspond to those for the single-server queue. That is, we assume an infinite population and an infinite queue size, with a single infinite queue shared among all servers. Unless otherwise stated, the dispatching discipline is FIFO. For the multiserver case, if all servers are assumed identical, the selection of a particular server for a waiting item has no effect on service time.

By way of contrast, Figure H.3b shows the structure of multiple single-server queues.

H.4 POISSON ARRIVAL RATE

Typically, analytic queueing models assume the arrival rate obeys a Poisson distribution. This is what is assumed in the results of Table 9.6. We define this distribution as follows. If items arrive at a queue according to a Poisson distribution, this may be expressed as

$$\Pr[k \text{ items arrive in time interval } T] = \frac{(\lambda T)^k}{k!} e^{-\lambda T}$$

$$\mathrm{E}[\text{number of items to arrive in time interval } T] = \lambda T$$

$$\text{Mean arrival rate, in items per second} = \lambda$$

Arrivals occurring according to a Poisson process are often referred to as **random arrivals**. This is because the probability of arrival of an item in a small interval is proportional to the length of the interval, and is independent of the amount of elapsed time since the arrival of the last item. That is, when items are arriving according to a Poisson process, an item is as likely to arrive at one instant as any other, regardless of the instants at which the other customers arrive.

Another interesting property of the Poisson process is its relationship to the exponential distribution. If we look at the times between arrivals of items T_a (called the interarrival times), then we find that this quantity obeys the exponential distribution:

$$\Pr[T_a < t] = 1 - e^{-\lambda t}$$

$$\mathrm{E}[T_a] = \frac{1}{\lambda}$$

Thus, the mean interarrival time is the reciprocal of the arrival rate, as we would expect.

APPENDIX I

THE COMPLEXITY OF ALGORITHMS

I.1 COMPLEXITY OVERVIEW

A central issue in assessing the practicality of an algorithm is the relative amount of time it takes to execute the algorithm. Typically, one cannot be sure that one has found the most efficient algorithm for a particular function. The most that one can say is that for a particular algorithm, the level of effort for execution is of a particular order of magnitude. One can then compare that order of magnitude to the speed of current or predicted processors to determine the level of practicality of a particular algorithm.

A common measure of the efficiency of an algorithm is its time complexity. We define the **time complexity** of an algorithm to be $f(n)$ if, for all n and all inputs of length n, the execution of the algorithm takes at most $f(n)$ steps. Thus, for a given size of input and a given processor speed, the time complexity is an upper bound on the execution time.

There are several ambiguities here. First, the definition of a step is not precise. A step could be a single operation of a Turing machine, a single processor machine instruction, a single high-level language machine instruction, and so on. However, these various definitions of step should all be related by simple multiplicative constants. For very large values of n, these constants are not important. What is important is how fast the relative execution time is growing.

A second issue is that, generally speaking, we cannot pin down an exact formula for $f(n)$. We can only approximate it. But again, we are primarily interested in the rate of change of $f(n)$ as n becomes very large.

There is a standard mathematical notation, known as the "big-O" notation, for characterizing the time complexity of algorithms that is useful in this context. The definition is as follows: $f(n) = O(g(n))$ if and only if there exist two numbers a and M such that

$$|f(n)| \leq a \times |g(n)|, \qquad n \geq M \tag{I.1}$$

An example helps clarify the use of this notation. Suppose we wish to evaluate a general polynomial of the form:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Consider the following simple-minded algorithm from [POHL81]:

```
algorithm P1;
    n, i, j: integer; x, polyval: real;
    a, S: array [0..100] of real;
    begin
     read(x, n);
     for i := 0 upto n do
     begin
          S[i] := 1; read(a[i]);
          for j := 1 upto i do S[i] := x × S[i];
          S[i] := a[i] × S[i]
     end;
```

```
    polyval := 0;
    for i := 0 upto n do polyval := polyval + S[i];
    write ('value at', x, 'is', polyval)
  end.
```

In this algorithm, each sub-expression is evaluated separately. Each S[i] requires (i + 1) multiplications: i multiplications to compute S[i] and one to multiply by a[i]. Computing all n terms requires

$$\sum_{i=0}^{n}(i+1) = \frac{(n+2)(n+1)}{2}$$

multiplications. There are also (n + 1) additions, which we can ignore relative to the much larger number of multiplications. Thus, the time complexity of this algorithm is $f(n) = (n+2)(n+1)/2$. We now show $f(n) = O(n^2)$. From the definition of Equation (I.1), we want to show that for $a = 1$ and $M = 4$, the relationship holds for $g(n) = n^2$. We do this by induction on n. The relationship holds for $n = 4$ because $(4+2)(4+1)/2 = 15 < 4^2 = 16$. Now assume it holds for all values of n up to k [i.e., $(k+2)(k+1)/2 < k^2$]. Then, with $n = k + 1$:

$$\begin{aligned}\frac{(n+2)(n+1)}{2} &= \frac{(k+3)(k+2)}{2}\\ &= \frac{(k+2)(k+1)}{2} + k + 2\\ &\le k^2 + k + 2\\ &\le k^2 + 2k + 1 = (k+1)^2 = n^2\end{aligned}$$

Therefore, the result is true for $n = k + 1$.

In general, the big-O notation makes use of the term that grows the fastest. For example:

1. $O[ax^7 + 3x^3 + \sin(x)] = O(ax^7) = O(x^7)$
2. $O(e^n + an^{10}) = O(e^n)$
3. $O(n! + n^{50}) = O(n!)$

There is much more to the big-O notation, with fascinating ramifications. For the interested reader, two of the best accounts are in [GRAH94] and [KNUT97].

An algorithm with an input of size n is said to be:

1. **Linear:** if the running time is $O(n)$
2. **Polynomial:** if the running time is $O(n^t)$ for some constant t
3. **Exponential:** if the running time is $O(t^{h(n)})$ for some constant t and polynomial $h(n)$

Generally, a problem that can be solved in polynomial time is considered feasible, whereas anything larger than polynomial time, especially exponential time, is considered infeasible. But you must be careful with these terms. First, if the size of the input is small enough, even very complex algorithms become feasible. Suppose, for example, you have a system that can execute 10^{12} operations per unit time.

Table I.1 Level of Effort for Various Levels of Complexity

Complexity	Size of Input	Operations
$\log_2 n$	$2^{10^{12}} = 10^{3\times10^{11}}$	10^{12}
n	10^{12}	10^{12}
n^2	10^6	10^{12}
n^6	10^2	10^{12}
2^n	39	10^{12}
$n!$	15	10^{12}

Table I-1 shows the size of input that can be handled in one time unit for algorithms of various complexities. For algorithms of exponential or factorial time, only very small inputs can be accommodated.

The second thing to be careful about is the way in which the input is characterized. For example, the complexity of cryptanalysis of an encryption algorithm can be characterized equally well in terms of the number of possible keys or the length of the key. For the Advanced Encryption Standard (AES), for example, the number of possible keys is 2^{128}, and the length of the key is 128 bits. If we consider a single encryption to be a "step" and the number of possible keys to be $N = 2^n$, then the time complexity of the algorithm is linear in terms of the number of keys [O(N)] but exponential in terms of the length of the key [O(2^n)].

I.2 REFERENCES

GRAH94 Graham, R.; Knuth, D.; and Patashnik, O. *Concrete Mathematics: A Foundation for Computer Science.* Reading, MA: Addison-Wesley, 1994.

KNUT97 Knuth, D. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Reading, MA: Addison-Wesley, 1997.

POHL81 Pohl, I., and Shaw, A. *The Nature of Computation: An Introduction to Computer Science.* Rockville, MD: Computer Science Press, 1981.

APPENDIX J

DISK STORAGE DEVICES

J.1 MAGNETIC DISK

A disk is a circular platter constructed of metal or of plastic coated with a magnetizable material. Data are recorded on and later retrieved from the disk via a conducting coil named the **head**. During a read or write operation, the head is stationary while the platter rotates beneath it.

The write mechanism exploits the fact that electricity flowing through a coil produces a magnetic field. Electric pulses are sent to the head, and magnetic patterns are recorded on the surface below, with different patterns for positive and negative currents. The read mechanism is based on the fact that a magnetic field moving relative to a coil produces an electrical current in the coil. When the surface of the disk passes under the head, it generates a current of the same polarity as the one already recorded.

Data Organization and Formatting

The head is a relatively small device capable of reading from or writing to a portion of the platter rotating beneath it. This gives rise to the organization of data on the platter in a concentric set of rings, called **tracks**. Each track is the same width as the head. There are thousands of tracks per surface.

Figure J.1 depicts this data layout. Adjacent tracks are separated by **gaps**. This prevents, or at least minimizes, errors due to misalignment of the head or simply interference of magnetic fields.

Data are transferred to and from the disk in **sectors** (see Figure J.1). There are typically hundreds of sectors per track, and these may be of either fixed or variable length. In most contemporary systems, fixed-length sectors are used, with 512 bytes being the nearly universal sector size. To avoid imposing unreasonable precision requirements on the system, adjacent sectors are separated by intratrack (intersector) gaps.

A bit near the center of a rotating disk travels past a fixed point (such as a read-write head) slower than a bit on the outside. Therefore, some way must be found to compensate for the variation in speed so the head can read all the bits at the same rate. This can be done by increasing the spacing between bits of information recorded in segments of the disk. The information can then be scanned at the same rate by rotating the disk at a fixed speed, known as the **constant angular velocity (CAV)**.

Figure J.2a shows the layout of a disk using CAV. The disk is divided into a number of pie-shaped sectors and into a series of concentric tracks. The advantage of using CAV is that individual blocks of data can be directly addressed by track and sector. To move the head from its current location to a specific address, it only takes a short movement of the head to a specific track and a short wait for the proper sector to spin under the head. The disadvantage of CAV is that the amount of data that can be stored on the long outer tracks is the same as what can be stored on the short inner tracks.

Because the **density**, in bits per linear inch, increases in moving from the outermost track to the innermost track, disk storage capacity in a straightforward CAV system is limited by the maximum recording density that can be achieved on the innermost track. To increase density, modern hard disk systems use a technique known as **multiple zone recording**, in which the surface is divided into a number of concentric zones (16 is typical). Within a zone, the number of bits per track is

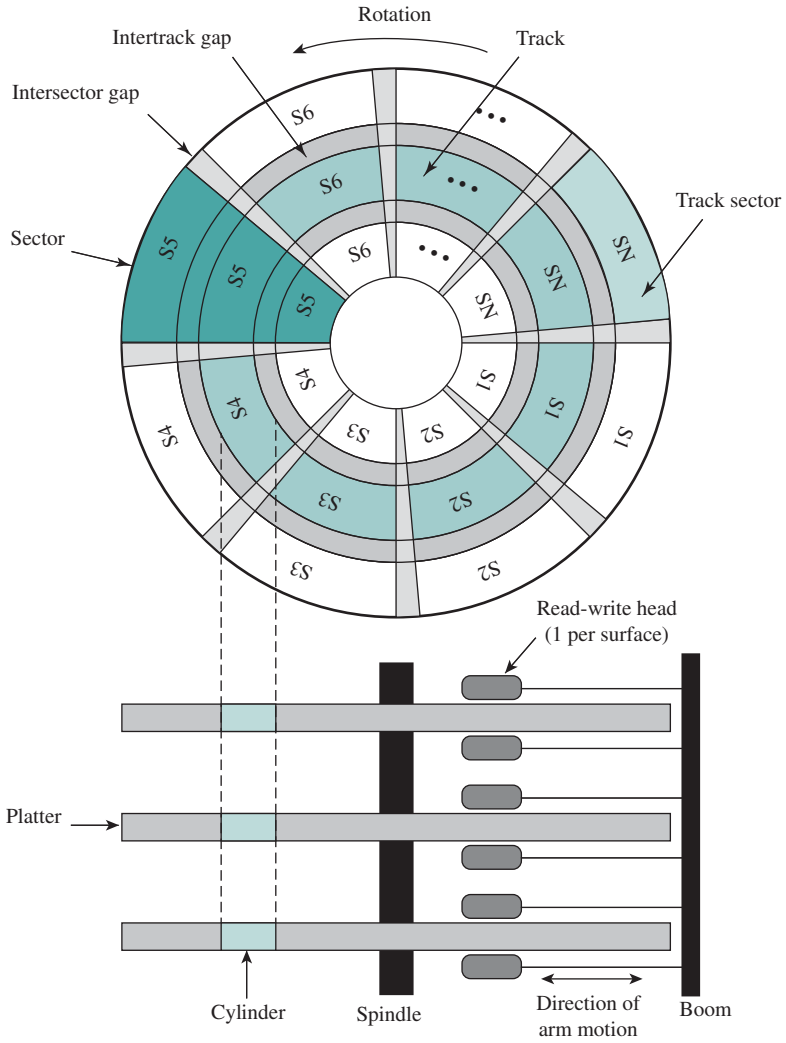
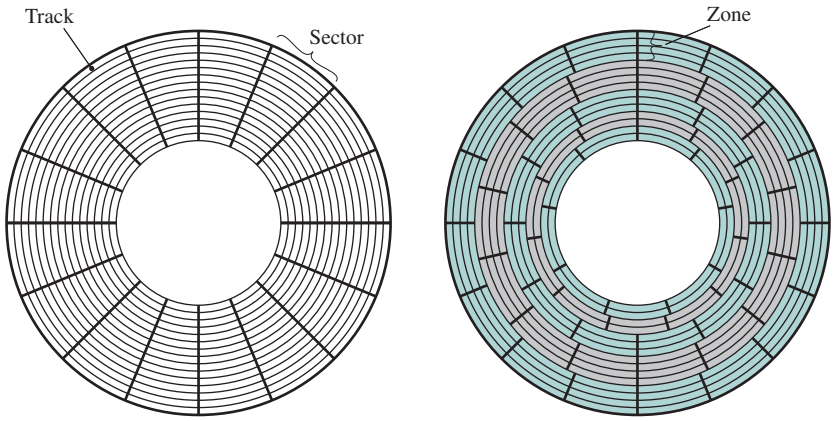


Figure J.1 Disk Data Layout

constant. Zones farther from the center contain more bits (more sectors) than zones closer to the center. This allows for greater overall storage capacity at the expense of somewhat more complex circuitry. As the disk head moves from one zone to another, the length (along the track) of individual bits changes, causing a change in the timing for reads and writes. Figure J.2b suggests the nature of multiple zone recording; in this illustration, each zone is only a single track wide.

Some means is needed to locate sector positions within a track. Clearly, there must be some starting point on the track, and a way of identifying the start and end of each sector. These requirements are handled by means of control data recorded on the disk. Thus, the disk is formatted with some extra data used only by the disk drive and not accessible to the user.



(a) Constant angular velocity (b) Multiple zone recording

Figure J.2 Comparison of Disk Layout Methods

Physical Characteristics

Table J.1 lists the major characteristics that differentiate among the various types of magnetic disks. First, the head may either be fixed or movable with respect to the radial direction of the platter. In a **fixed-head disk**, there is one read/write head per track. All of the heads are mounted on a rigid arm that extends across all tracks; such systems are rare today. In a **movable-head disk**, there is only one read/write head. Again, the head is mounted on an arm. Because the head must be able to be positioned above any track, the arm can be extended or retracted for this purpose.

The disk itself is mounted in a disk drive, which consists of the arm, a spindle that rotates the disk, and the electronics needed for input and output of binary data. A **nonremovable disk** is permanently mounted in the disk drive; the hard disk in a personal computer is a nonremovable disk. A **removable disk** can be removed and replaced with another disk. The advantage of the latter type is that unlimited amounts of data are available with a limited number of disk systems. Furthermore, such a disk may be moved from one computer system to another.

For most disks, the magnetizable coating is applied to both sides of the platter, which is then referred to as **double-sided**. Some less expensive disk systems use **single-sided** disks.

Table J.1 Physical Characteristics of Disk Systems

Head Motion	**Platters**
Fixed head (one per track)	Single platter
Movable head (one per surface)	Multiple platter
Disk Portability	**Head Mechanism**
Nonremovable disk	Contact (floppy)
Removable disk	Fixed gap
Sides	Aerodynamic gap (Winchester)
Single sided	
Double sided	

Some disk drives accommodate **multiple platters** stacked vertically a fraction of an inch apart. Multiple arms are provided (Figure J.3). Multiple-platter disks employ a movable head, with one read-write head per platter surface. All of the heads are mechanically fixed so all are at the same distance from the center of the disk and move together. Thus, at any time, all of the heads are positioned over tracks that are of equal distance from the center of the disk. The set of all the tracks in the same relative position on the platter is referred to as a **cylinder**. For example, all of the shaded tracks in Figure J.4 are part of one cylinder.

Finally, the head mechanism provides a classification of disks into three types. Traditionally, the read/write head has been positioned at a fixed distance above the platter, allowing an air gap. At the other extreme is a head mechanism that actually comes into physical contact with the medium during a read or write operation. This mechanism is used with the **floppy disk**, which is a small, flexible platter and the least expensive type of disk.

To understand the third type of disk, we need to comment on the relationship between data density and the size of the air gap. The head must generate or sense an electromagnetic field of sufficient magnitude to write and read properly. The narrower the head is, the closer it must be to the platter surface to function. A narrower head means narrower tracks and therefore greater data density, which is desirable.

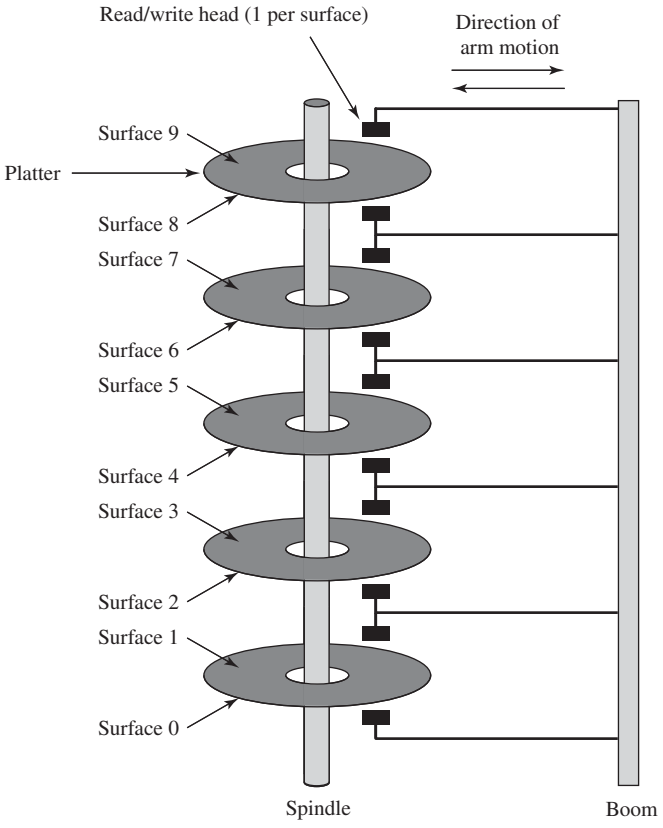


Figure J.3 Components of a Disk Drive

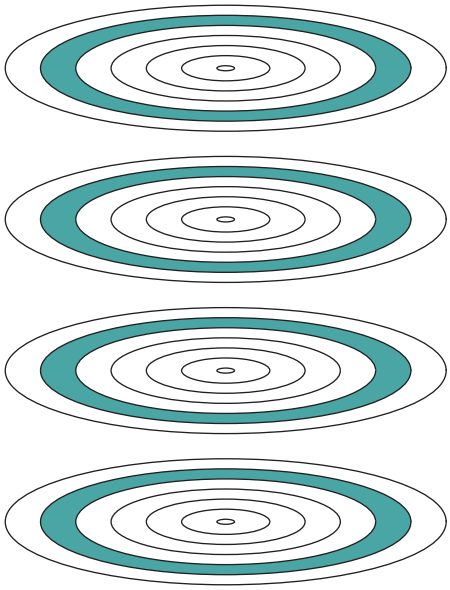


Figure J.4 Tracks and Cylinders

However, the closer the head is to the disk, the greater the risk of error from impurities or imperfections. To push the technology further, the **Winchester disk** was developed. Winchester heads are used in sealed drive assemblies that are almost free of contaminants. They are designed to operate closer to the disk's surface than conventional rigid disk heads, thus allowing greater data density. The head is actually an aerodynamic foil that rests lightly on the platter's surface when the disk is motionless. The air pressure generated by a spinning disk is enough to make the foil rise above the surface. The resulting noncontact system can be engineered to use narrower heads that operate closer to the platter's surface than conventional rigid disk heads.

Table J.2 gives disk parameters for typical contemporary high-performance disks.

J.2 OPTICAL MEMORY

In 1983, one of the most successful consumer products of all time was introduced: the compact disk (CD) digital audio system. The CD is a nonerasable disk that can store more than 60 minutes of audio information on one side. The huge commercial success of the CD enabled the development of low-cost optical-disk storage technology that has revolutionized computer data storage. A variety of optical-disk systems are in use (see Table J.3). We briefly review each of these.

CD-ROM

The audio CD and the CD-ROM (compact disk read-only memory) share a similar technology. The main difference is that CD-ROM players are more rugged and have error-correction devices to ensure data are properly transferred from disk to

Table J.2 Typical Hard Disk Drive Parameters

Characteristics	Seagate Barracuda ES.2	Seagate Barracuda 7200.10	Seagate Barracuda 7200.9	Seagate	Hitachi Microdrive
Application	High-capacity server	High-performance desktop	Entry-level desktop	Laptop	Handheld devices
Capacity	1 TB	750 GB	160 GB	120 GB	8 GB
Minimum track-to-track seek time	0.8 ms	0.3 ms	1.0 ms	—	1.0 ms
Average seek time	8.5 ms	3.6 ms	9.5 ms	12.5 ms	12 ms
Spindle speed	7200 rpm	7200 rpm	7200	5400 rpm	3600 rpm
Average rotational delay	4.16 ms	4.16 ms	4.17 ms	5.6 ms	8.33 ms
Maximum transfer rate	3 GB/s	300 MB/s	300 MB/s	150 MB/s	10 MB/s
Bytes per sector	512	512	512	512	512
Tracks per cylinder (number of platter surfaces)	8	8	2	8	2

computer. Both types of disk are made in the same way. The disk is formed from a resin, such as polycarbonate. Digitally recorded information (either music or computer data) is imprinted as a series of microscopic pits on the surface of the polycarbonate. This is done, first of all, with a finely focused, high-intensity laser to create a master disk. The master is used, in turn, to make a die to stamp out copies onto polycarbonate. The pitted surface is then coated with a highly reflective surface, usually aluminum or gold. This shiny surface is protected against dust and scratches by a top coat of clear acrylic. Finally, a label can be silkscreened onto the acrylic.

Information is retrieved from a CD or CD-ROM by a low-powered laser housed in an optical-disk player, or drive unit. The laser shines through the clear polycarbonate while a motor spins the disk past it (see Figure J.5). The intensity of the reflected light of the laser changes as it encounters a pit. Specifically, if the laser beam falls on a pit, which has a somewhat rough surface, the light scatters and a low intensity is reflected back to the source. The areas between pits are called *lands*. A land is a smooth surface, which reflects back at higher intensity. The change between pits and lands is detected by a photosensor and converted into a digital signal. The sensor tests the surface at regular intervals. The beginning or end of a pit represents a 1; when no change in elevation occurs between intervals, a 0 is recorded.

Recall that on a magnetic disk, information is recorded in concentric tracks. With the simplest CAV system, the number of bits per track is constant. An increase in density is achieved with multiple zoned recording, in which the surface is divided into a number of zones, with zones farther from the center containing more bits than zones closer to the center. Although this technique increases capacity, it is still not optimal.

To achieve greater capacity, CDs and CD-ROMs do not organize information on concentric tracks. Instead, the disk contains a single spiral track, beginning near

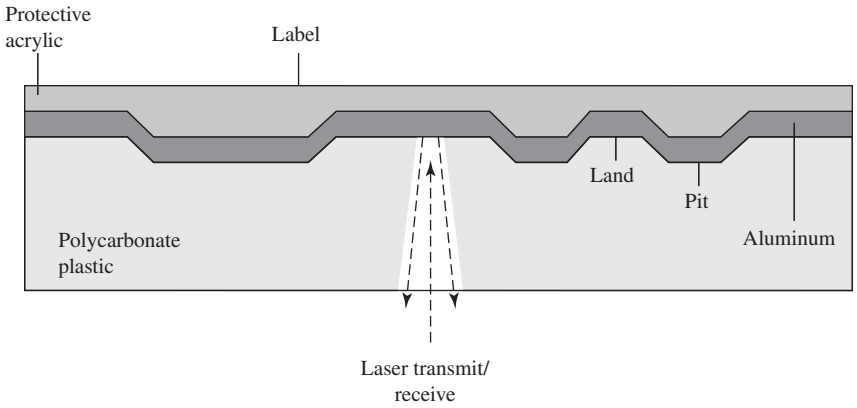


Figure J.5 CD Operation

the center and spiraling out to the outer edge of the disk. Sectors near the outside of the disk are the same length as those near the inside. Thus, information is packed evenly across the disk in segments of the same size, and these are scanned at the same rate by rotating the disk at a variable speed. The pits are then read by the laser at a **constant linear velocity (CLV)**. The disk rotates more slowly for accesses near the outer edge than for those near the center. Thus, the capacity of a track and the rotational delay both increase for positions nearer the outer edge of the disk. The data capacity for a CD-ROM is about 680 MB.

CD-ROM is appropriate for the distribution of large amounts of data to a large number of users. Because of the expense of the initial writing process, it is not appropriate for individualized applications. Compared with traditional magnetic disks, the CD-ROM has three major advantages:

- The information-storage capacity is much greater on the optical disk.
- The optical disk together with the information stored on it can be mass replicated inexpensively—unlike a magnetic disk. The data on a magnetic disk has to be reproduced by copying one disk at a time using two disk drives.
- The optical disk is removable, allowing the disk itself to be used for archival storage. Most magnetic disks are nonremovable. The information on nonremovable magnetic disks must first be copied to some other storage device before the disk drive/disk can be used to store new information.

The disadvantages of CD-ROM are as follows:

- It is read-only and cannot be updated.
- It has an access time much longer than that of a magnetic disk drive, as much as half a second.

CD Recordable

To accommodate applications in which only one or a small number of copies of a set of data is needed, the write-once read-many CD, known as the CD recordable (CD-R) has been developed. For CD-R, a disk is prepared in such a way that it can

be subsequently written once with a laser beam of modest intensity. Thus, with a somewhat more expensive disk controller than for CD-ROM, the customer can write once as well as read the disk.

The CD-R medium is similar to, but not identical to, that of a CD or CD-ROM. For CDs and CD-ROMs, information is recorded by the pitting of the surface of the medium, which changes reflectivity. For a CD-R, the medium includes a dye layer. The dye is used to change reflectivity and is activated by a high-intensity laser. The resulting disk can be read on a CD-R drive or a CD-ROM drive.

The CD-R optical disk is attractive for archival storage of documents and files. It provides a permanent record of large volumes of user data.

CD Rewritable

The CD-RW optical disk can be repeatedly written and overwritten, as with a magnetic disk. Although a number of approaches have been tried, the only pure optical approach that has proved attractive is called phase change. The phase change disk uses a material that has two significantly different reflectivities in two different phase states. There is an amorphous state, in which the molecules exhibit a random orientation that reflects light poorly; and a crystalline state, which has a smooth surface that reflects light well. A beam of laser light can change the material from one phase to the other. The primary disadvantage of phase change optical disks is that the material eventually and permanently loses its desirable properties. Current materials can be used for between 500,000 and 1,000,000 erase cycles.

The CD-RW has the obvious advantage over CD-ROM and CD-R that it can be rewritten and thus used as a true secondary storage. As such, it competes with magnetic disk. A key advantage of the optical disk is that the engineering tolerances for optical disks are much less severe than for high-capacity magnetic disks. Thus, optical disks exhibit higher reliability and longer life.

Digital Versatile Disk

With the capacious digital versatile disk (DVD), the electronics industry has at last found an acceptable replacement for the analog VHS video tape. The DVD will replace the video tape used in video cassette recorders (VCRs) and, more important for this discussion, replace the CD-ROM in personal computers and servers. The DVD takes video into the digital age. It delivers movies with impressive picture quality, and it can be randomly accessed like audio CDs, which DVD machines can also play. Vast volumes of data can be crammed onto the disk, currently seven times as much as a CD-ROM. With DVD's huge storage capacity and vivid quality, PC games will become more realistic, and educational software will incorporate more video. Following in the wake of these developments will be a new crest of traffic over the Internet and corporate intranets, as this material is incorporated into websites.

The DVD's greater capacity is due to three differences from CDs:

1. Bits are packed more closely on a DVD. The spacing between loops of a spiral on a CD is 1.6 μm and the minimum distance between pits along the spiral is 0.834 μm. The DVD uses a laser with shorter wavelength and achieves a loop

spacing of 0.74 μm and a minimum distance between pits of 0.4 μm. The result of these two improvements is about a sevenfold increase in capacity, to about 4.7 GB.

2. The DVD employs a second layer of pits and lands on top of the first layer. A dual-layer DVD has a semireflective layer on top of the reflective layer, and by adjusting focus, the lasers in DVD drives can read each layer separately. This technique almost doubles the capacity of the disk, to about 8.5 GB. The lower reflectivity of the second layer limits its storage capacity so a full doubling is not achieved.
3. The DVD-ROM can be two sided, whereas data are recorded on only one side of a CD. This brings total capacity up to 17 GB.

As with the CD, DVDs come in writeable as well as read-only versions (see Table J.3).

High-Definition Optical Disks

High-definition optical disks are designed to store high-definition videos and to provide significantly greater storage capacity compared to DVDs. The higher bit density is achieved by using a laser with a shorter wavelength, in the blue-violet range. The data pits, which constitute the digital 1s and 0s, are smaller on the high-definition optical disks compared to DVD because of the shorter laser wavelength.

Table J.3 Optical Disk Products

CD
Compact Disk. A nonerasable disk that stores digitized audio information. The standard system uses 12-cm disks and can record more than 60 minutes of uninterrupted playing time.
CD-ROM
Compact Disk Read-Only Memory. A nonerasable disk used for storing computer data. The standard system uses 12-cm disks and can hold more than 650 Mbytes.
CD-R
CD Recordable. Similar to a CD-ROM. The user can write to the disk only once.
CD-RW
CD Rewritable. Similar to a CD-ROM. The user can erase and rewrite to the disk multiple times.
DVD
Digital Versatile Disk. A technology for producing digitized, compressed representation of video information, as well as large volumes of other digital data. Both 8- and 12-cm diameters are used, with a double-sided capacity of up to 17 GB. The basic DVD is read-only (DVD-ROM).
DVD-R
DVD Recordable. Similar to a DVD-ROM. The user can write to the disk only once. Only one-sided disks can be used.
DVD-RW
DVD Rewritable. Similar to a DVD-ROM. The user can erase and rewrite to the disk multiple times. Only one-sided disks can be used.
Blu-Ray DVD
High definition video disk. Provides considerably greater data storage density than DVD, using a 405-nm (blue-violet) laser. A single layer on a single side can store 25 GB.

Two disk formats and technologies initially competed for market acceptance: HD DVD and Blu-ray DVD. The Blu-ray scheme ultimately achieved market dominance. The HD DVD scheme can store 15 GB on a single layer on a single side. Blu-ray positions the data layer on the disk closer to the laser (shown on the right-hand side of each diagram in Figure J.6). This enables a tighter focus and less distortion and thus smaller pits and tracks. Blu-ray can store 25 GB on a single layer. Three versions are available: read only (BD-ROM), recordable once (BD-R), and rerecordable (BD-RE).

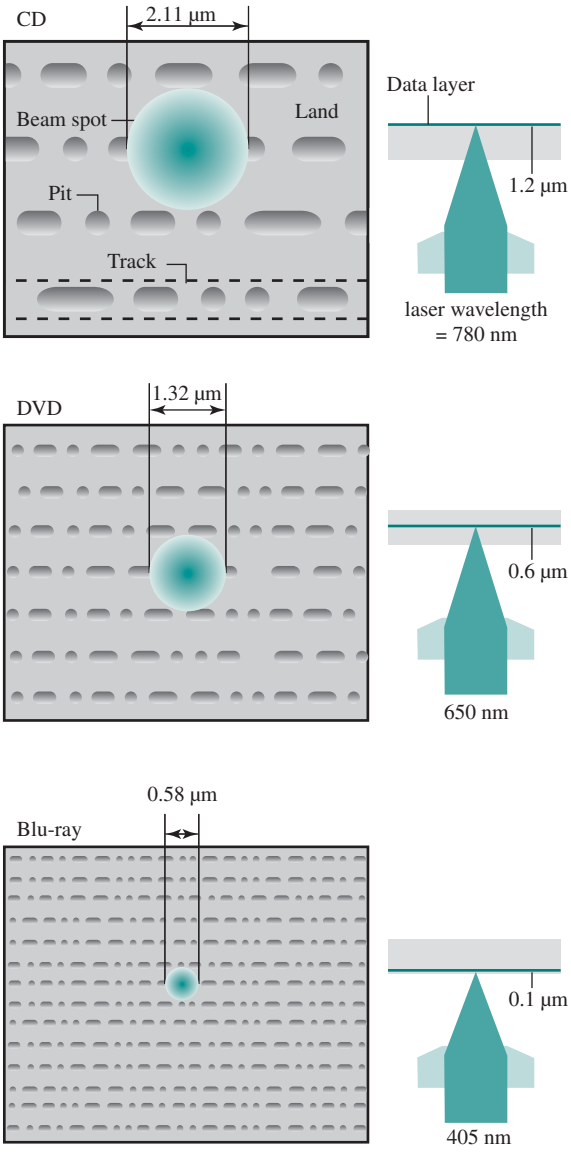


Figure J.6 **Optical Memory Characteristics**

APPENDIX K

CRYPTOGRAPHIC ALGORITHMS

The essential technology underlying virtually all automated network and computer security applications is cryptography. Two fundamental approaches are in use: symmetric encryption, also known as conventional encryption, and public-key encryption, also known as asymmetric encryption. This appendix provides an overview of both types of encryption, together with a brief discussion of some important encryption algorithms.

K.1 SYMMETRIC ENCRYPTION

Symmetric encryption was the only type of encryption in use prior to the introduction of public-key encryption in the late 1970s. Symmetric encryption has been used for secret communication by countless individuals and groups, from Julius Caesar to the German U-boat force to present-day diplomatic, military, and commercial users. It remains by far the more widely used of the two types of encryption.

A symmetric encryption scheme has five ingredients (see Figure K.1):

- **Plaintext:** This is the original message or data that is fed into the algorithm as input.
- **Encryption algorithm:** The encryption algorithm performs various substitutions and transformations on the plaintext.
- **Secret key:** The secret key is also input to the encryption algorithm. The exact substitutions and transformations performed by the algorithm depend on the key.
- **Ciphertext:** This is the scrambled message produced as output. It depends on the plaintext and the secret key. For a given message, two different keys will produce two different ciphertexts.
- **Decryption algorithm:** This is essentially the encryption algorithm run in reverse. It takes the ciphertext and the secret key and produces the original plaintext.

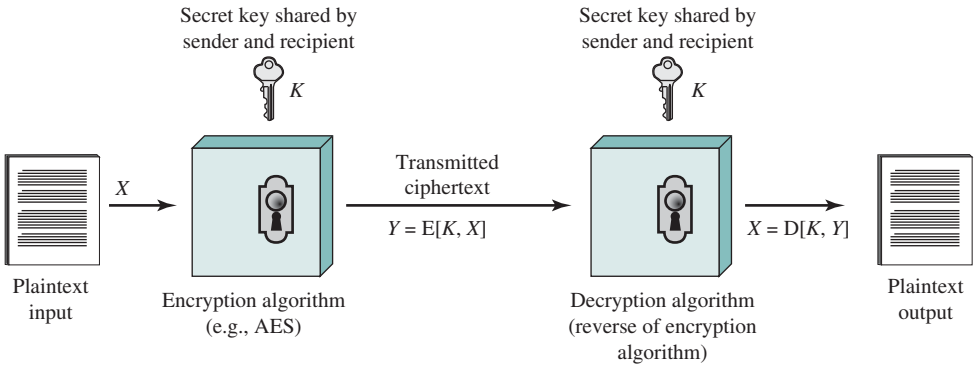


Figure K.1 Simplified Model of Symmetric Encryption

There are two requirements for secure use of symmetric encryption:

1. We need a strong encryption algorithm. At a minimum, we would like the algorithm to be such that an opponent who knows the algorithm and has access to one or more ciphertexts would be unable to decipher the ciphertext or figure out the key. This requirement is usually stated in a stronger form: The opponent should be unable to decrypt ciphertext or discover the key, even if he or she is in possession of a number of ciphertexts together with the plaintext that produced each ciphertext.
2. Sender and receiver must have obtained copies of the secret key in a secure fashion and must keep the key secure. If someone can discover the key and knows the algorithm, all communication using this key is readable.

There are two general approaches to attacking a symmetric encryption scheme. The first attack is known as **cryptanalysis**. Cryptanalytic attacks rely on the nature of the algorithm plus perhaps some knowledge of the general characteristics of the plaintext or even some sample plaintext-ciphertext pairs. This type of attack exploits the characteristics of the algorithm to attempt to deduce a specific plaintext or to deduce the key being used. If the attack succeeds in deducing the key, the effect is catastrophic: All future and past messages encrypted with that key are compromised.

The second method, known as the **brute-force** attack, is to try every possible key on a piece of ciphertext until an intelligible translation into plaintext is obtained. On average, half of all possible keys must be tried to achieve success. Table K.1 shows how much time is involved for various key sizes. The table shows results for each key size, assuming it takes 1 μs to perform a single decryption, a reasonable order of magnitude for today's computers. With the use of massively parallel organizations of microprocessors, it may be possible to achieve processing rates many orders of magnitude greater. The final column of the table considers the results for a system that can process 1 million keys per microsecond. As one can see, at this performance level, a 56-bit key can no longer be considered computationally secure.

The most commonly used symmetric encryption algorithms are block ciphers. A block cipher processes the plaintext input in fixed-size blocks and produces a block of ciphertext of equal size for each plaintext block. The two most important symmetric algorithms, both of which are block ciphers, are the Data Encryption Standard (DES) and the Advanced Encryption Standard (AES).

Table K.1 Average Time Required for Exhaustive Key Search

Key Size (bits)	Number of Alternative Keys	Time Required at 1 Decryption/μs	Time Required at 10^6 Decryptions/μs
32	$2^{32} = 4.3 \times 10^9$	$2^{31}\ \mu s = 35.8$ minutes	2.15 milliseconds
56	$2^{56} = 7.2 \times 10^{16}$	$2^{55}\ \mu s = 1142$ years	10.01 hours
128	$2^{128} = 3.4 \times 10^{38}$	$2^{127}\ \mu s = 5.4 \times 10^{24}$ years	5.4×10^{18} years
168	$2^{168} = 3.7 \times 10^{50}$	$2^{167}\ \mu s = 5.9 \times 10^{36}$ years	5.9×10^{30} years
26 characters (permutation)	$26! = 4 \times 10^{26}$	$2 \times 10^{26}\ \mu s = 6.4 \times 10^{12}$ years	6.4×10^6 years

The Data Encryption Standard (DES)

DES has been the dominant encryption algorithm since its introduction in 1977. However, because DES uses only a 56-bit key, it was only a matter of time before computer processing speed made DES obsolete. In 1998, the Electronic Frontier Foundation (EFF) announced that it had broken a DES challenge using a special-purpose "DES cracker" machine that was built for less than $250,000. The attack took less than three days. The EFF has published a detailed description of the machine, enabling others to build their own cracker. And, of course, hardware prices continue to drop as speeds increase, making DES worthless.

The life of DES was extended by the use of triple DES (3DES), which involves repeating the basic DES algorithm three times, using either two or three unique keys, for a key size of 112 or 168 bits.

The principal drawback of 3DES is that the algorithm is relatively sluggish in software. A secondary drawback is that both DES and 3DES use a 64-bit block size. For reasons of both efficiency and security, a larger block size is desirable.

Advanced Encryption Standard

Because of these drawbacks, 3DES is not a reasonable candidate for long-term use. As a replacement, the National Institute of Standards and Technology (NIST) in 1997 issued a call for proposals for a new Advanced Encryption Standard (AES), which should have a security strength equal to or better than 3DES and significantly improved efficiency. In addition to these general requirements, NIST specified that AES must be a symmetric block cipher with a block length of 128 bits and support for key lengths of 128, 192, and 256 bits. Evaluation criteria include security, computational efficiency, memory requirements, hardware and software suitability, and flexibility. In 2001, NIST issued AES as a federal information processing standard (FIPS 197).

K.2 PUBLIC-KEY CRYPTOGRAPHY

Public-key encryption, first publicly proposed by Diffie and Hellman in 1976, is the first truly revolutionary advance in encryption in literally thousands of years. For one thing, public-key algorithms are based on mathematical functions rather than on simple operations on bit patterns. More important, public-key cryptography is asymmetric, involving the use of two separate keys, in contrast to symmetric encryption, which uses only one key. The use of two keys has profound consequences in the areas of confidentiality, key distribution, and authentication.

Before proceeding, we should first mention several common misconceptions concerning public-key encryption. One is that public-key encryption is more secure from cryptanalysis than symmetric encryption. In fact, the security of any encryption scheme depends on the length of the key and the computational work involved in breaking a cipher. There is nothing in principle about either symmetric or public-key encryption that makes one superior to another from the point of view of resisting cryptanalysis. A second misconception is that public-key encryption is a general-purpose technique that has made symmetric encryption obsolete. On the contrary, because of the computational overhead of current public-key encryption schemes,

there seems no foreseeable likelihood that symmetric encryption will be abandoned. Finally, there is a feeling that key distribution is trivial when using public-key encryption, compared to the rather cumbersome handshaking involved with key distribution centers for symmetric encryption. In fact, some form of protocol is needed, often involving a central agent, and the procedures involved are no simpler or any more efficient than those required for symmetric encryption.

A public-key encryption scheme has six ingredients (see Figure K.2):

- **Plaintext:** This is the readable message or data that is fed into the algorithm as input.
- **Encryption algorithm:** The encryption algorithm performs various transformations on the plaintext.
- **Public and private key:** This is a pair of keys that have been selected so if one is used for encryption, the other is used for decryption. The exact transformations performed by the encryption algorithm depend on the public or private key that is provided as input.
- **Ciphertext:** This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.
- **Decryption algorithm:** This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

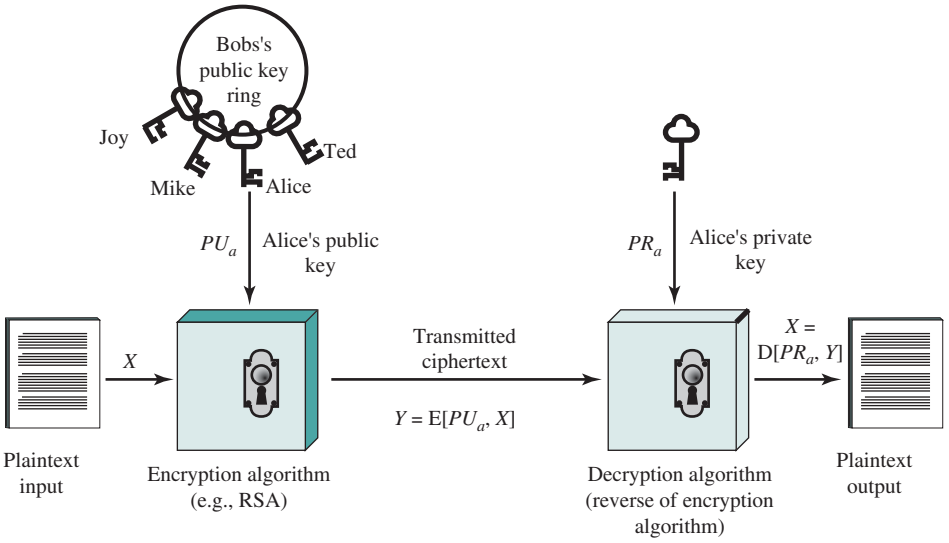
The process works (produces the correct plaintext on output) regardless of the order in which the pair of keys is used. As the names suggest, the public key of the pair is made public for others to use, while the private key is known only to its owner.

Now, say Bob wants to send a private message to Alice, and suppose that he has Alice's public key and Alice has the matching private key (see Figure K.2a). Using Alice's public key, Bob encrypts the message to produce ciphertext. The ciphertext is then transmitted to Alice. When Alice gets the ciphertext, she decrypts it using her private key. Because only Alice has a copy of her private key, no one else can read the message.

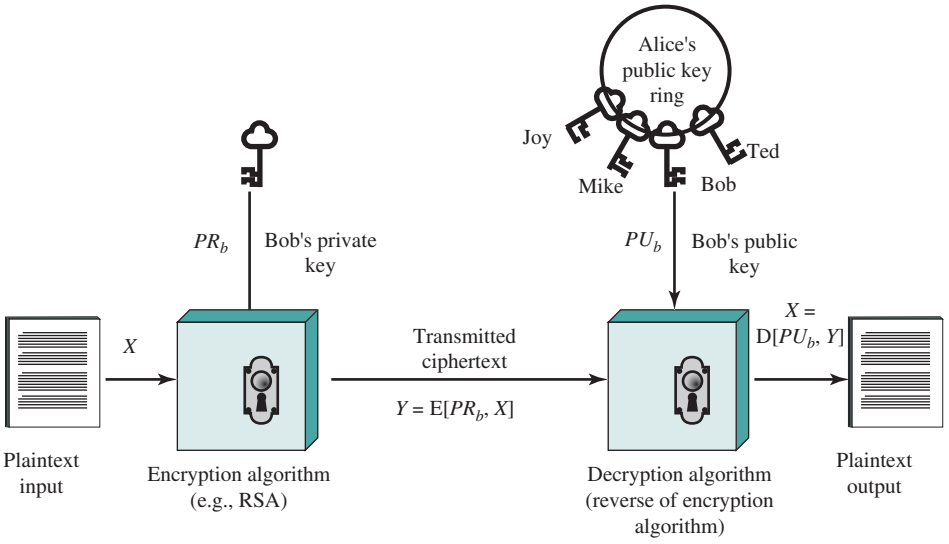
Public-key encryption can be used in another way, as illustrated in Figure K.2b. Suppose Bob wants to send a message to Alice and, although it isn't important that the message be kept secret, he wants Alice to be certain that the message is indeed from him. In this case Bob uses his own private key to encrypt the message. When Alice receives the ciphertext, she finds that she can decrypt it with Bob's public key, thus proving that the message must have been encrypted by Bob: No one else has Bob's private key, and therefore no one else could have created a ciphertext that could be decrypted with Bob's public key.

A general-purpose public-key cryptographic algorithm relies on one key for encryption and a different but related key for decryption. Furthermore, these algorithms have the following important characteristics:

- It is computationally infeasible to determine the decryption key given only knowledge of the cryptographic algorithm and the encryption key.
- Either of the two related keys can be used for encryption, with the other used for decryption.



(a) Encryption with public key



(b) Encryption with private key

Figure K.2 Public-Key Cryptography

The essential steps are the following:

1. Each user generates a pair of keys to be used for the encryption and decryption of messages.
2. Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private. As Figure K.2a suggests, each user maintains a collection of public keys obtained from others.

3. If Bob wishes to send a private message to Alice, Bob encrypts the message using Alice's public key.
4. When Alice receives the message, she decrypts it using her private key. No other recipient can decrypt the message because only Alice knows her own private key.

With this approach, all participants have access to public keys, and private keys are generated locally by each participant and therefore need never be distributed. As long as a user protects his or her private key, incoming communication is secure. At any time, a user can change the private key and publish the companion public key to replace the old public key.

The key used in symmetric encryption is typically referred to as a **secret key**. The two keys used for public-key encryption are referred to as the **public key** and the **private key**. Invariably, the private key is kept secret, but it is referred to as a private key rather than a secret key to avoid confusion with symmetric encryption.

Rivest-Shamir-Adleman (RSA) Algorithm

One of the first public-key schemes was developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman at MIT. The RSA scheme has since that time reigned supreme as the only widely accepted and implemented approach to public-key encryption. RSA is a cipher in which the plaintext and ciphertext are integers between 0 and $n - 1$ for some n. Encryption involves modular arithmetic. The strength of the algorithm is based on the difficulty of factoring numbers into their prime factors.

K.3 MESSAGE AUTHENTICATION AND HASH FUNCTIONS

Encryption protects against passive attack (eavesdropping). A different requirement is to protect against active attack (falsification of data and transactions). Protection against such attacks is known as message authentication.

A message, file, document, or other collection of data is said to be authentic when it is genuine and came from its alleged source. Message authentication is a procedure that allows communicating parties to verify that received messages are authentic. The two important aspects are to verify that the contents of the message have not been altered, and that the source is authentic. We may also wish to verify a message's timeliness (it has not been artificially delayed and replayed) and sequence relative to other messages flowing between two parties.

Authentication Using Symmetric Encryption

It is possible to perform authentication simply by the use of symmetric encryption. If we assume only the sender and receiver share a key (which is as it should be), then only the genuine sender would be able successfully to encrypt a message for the other participant. Furthermore, if the message includes an error-detection code and a sequence number, the receiver is assured no alterations have been made and sequencing is proper. If the message also includes a timestamp, the receiver is assured that the message has not been delayed beyond that normally expected for network transit.

Message Authentication without Message Encryption

In this section, we examine several approaches to message authentication that do not rely on message encryption. In all of these approaches, an authentication tag is generated and appended to each message for transmission. The message itself is not encrypted and can be read at the destination independent of the authentication function at the destination.

Because the approaches discussed in this section do not encrypt the message, message confidentiality is not provided. Because symmetric encryption will provide authentication, and because it is widely used with readily available products, why not simply use such an approach, which provides both confidentiality and authentication? Here are three situations in which message authentication without confidentiality is preferable:

1. There are a number of applications in which the same message is broadcast to a number of destinations. An example is the notification to users that the network is now unavailable or an alarm signal in a control center. It is cheaper and more reliable to have only one destination responsible for monitoring authenticity. Thus, the message must be broadcast in plaintext with an associated message authentication tag. The responsible system performs authentication. If a violation occurs, the other destination systems are alerted by a general alarm.
2. Another possible scenario is an exchange in which one side has a heavy load and cannot afford the time to decrypt all incoming messages. Authentication is carried out on a selective basis, with messages chosen at random for checking.
3. Authentication of a computer program in plaintext is an attractive service. The computer program can be executed without having to decrypt it every time, which would be wasteful of processor resources. However, if a message authentication tag were attached to the program, it could be checked whenever assurance is required of the integrity of the program.

Thus, there is a place for both authentication and encryption in meeting security requirements.

Message Authentication Code

One authentication technique involves the use of a secret key to generate a small block of data, known as a message authentication code, that is appended to the message. This technique assumes that two communicating parties, say A and B, share a common secret key K_{AB}. When A has a message M to send to B, it calculates the message authentication code as a function of the message and the key: $\text{MAC}_M = \text{F}(K_{AB}, M)$. The message plus code are transmitted to the intended recipient. The recipient performs the same calculation on the received message, using the same secret key, to generate a new message authentication code. The received code is compared to the calculated code (see Figure K.3). If we assume only the receiver

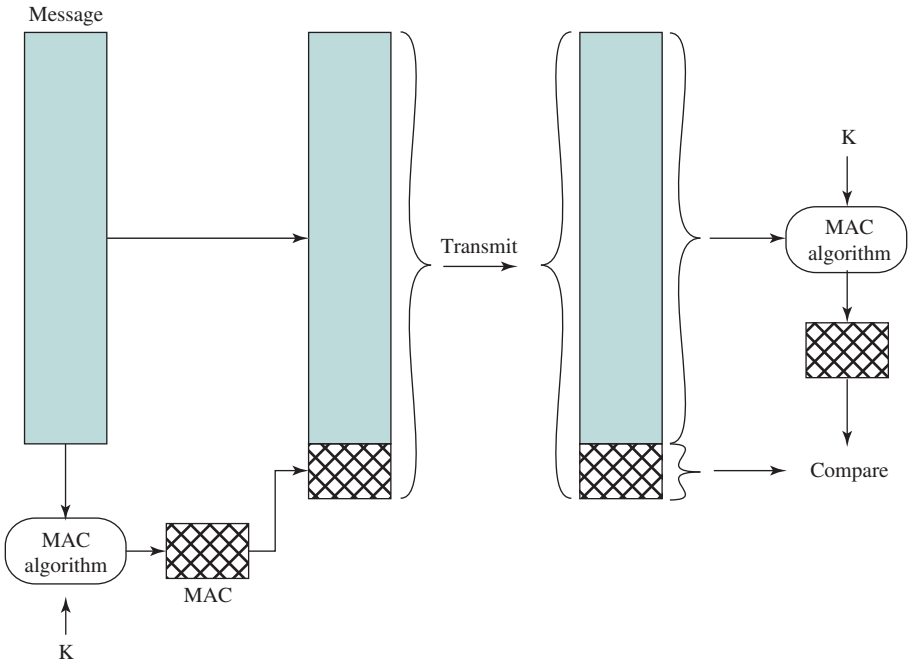


Figure K.3 Message Authentication Using a Message Authentication Code (MAC)

and the sender know the identity of the secret key, and if the received code matches the calculated code, then:

1. The receiver is assured the message has not been altered. If an attacker alters the message but does not alter the code, then the receiver's calculation of the code will differ from the received code. Because the attacker is assumed not to know the secret key, the attacker cannot alter the code to correspond to the alterations in the message.
2. The receiver is assured the message is from the alleged sender. Because no one else knows the secret key, no one else could prepare a message with a proper code.
3. If the message includes a sequence number (such as is used with X.25, HDLC, and TCP), then the receiver can be assured of the proper sequence, because an attacker cannot successfully alter the sequence number.

A number of algorithms could be used to generate the code. The National Bureau of Standards, in its publication *DES Modes of Operation*, recommends the use of DES. DES is used to generate an encrypted version of the message, and the last number of bits of ciphertext are used as the code. A 16- or 32-bit code is typical.

The process just described is similar to encryption. One difference is that the authentication algorithm need not be reversible, as it must for decryption. It turns out that because of the mathematical properties of the authentication function, it is less vulnerable to being broken than encryption.

One-Way Hash Function

A variation on the message authentication code that has received much attention is the one-way hash function. As with the message authentication code, a hash function accepts a variable-size message M as input and produces a fixed-size message digest H(M) as output. Unlike the MAC, a hash function does not also take a secret key as input. To authenticate a message, the message digest is sent with the message in such a way that the message digest is authentic.

Figure K.4 illustrates three ways in which the message can be authenticated. The message digest can be encrypted using symmetric encryption (part a); if it is assumed only the sender and receiver share the encryption key, then authenticity is assured. The message digest can also be encrypted using public-key encryption (part b). The public-key approach has two advantages: it provides a digital signature as well as message authentication, and it does not require the distribution of keys to communicating parties.

These two approaches have an advantage over approaches that encrypt the entire message in that less computation is required. Nevertheless, there has been interest in developing a technique that avoids encryption altogether. Several reasons for this interest are as follows:

- Encryption software is somewhat slow. Even though the amount of data to be encrypted per message is small, there may be a steady stream of messages into and out of a system.
- Encryption hardware costs are nonnegligible. Low-cost chip implementations of DES are available, but the cost adds up if all nodes in a network must have this capability.
- Encryption hardware is optimized toward large data sizes. For small blocks of data, a high proportion of the time is spent in initialization/invocation overhead.
- Encryption algorithms may be covered by patents and must be licensed, adding a cost.
- Encryption algorithms may be subject to export control.

Figure K.4c shows a technique that uses a hash function but no encryption for message authentication. This technique assumes that two communicating parties, say A and B, share a common secret value S_{AB}. When A has a message to send to B, it calculates the hash function over the concatenation of the secret value and the message: $MD_M = H(S_{AB}||M)$.[1] It then sends $[M||MD_M]$ to B. Because B possesses S_{AB}, it can recompute $H(S_{AB}||M)$ and verify MD_M. Because the secret value itself is not sent, it is not possible for an attacker to modify an intercepted message. As long as the secret value remains secret, it is also not possible for an attacker to generate a false message.

This third technique, using a shared secret value, is the one adopted for IP security; it has also been specified for SNMPv3.

[1] || denotes concatenation.

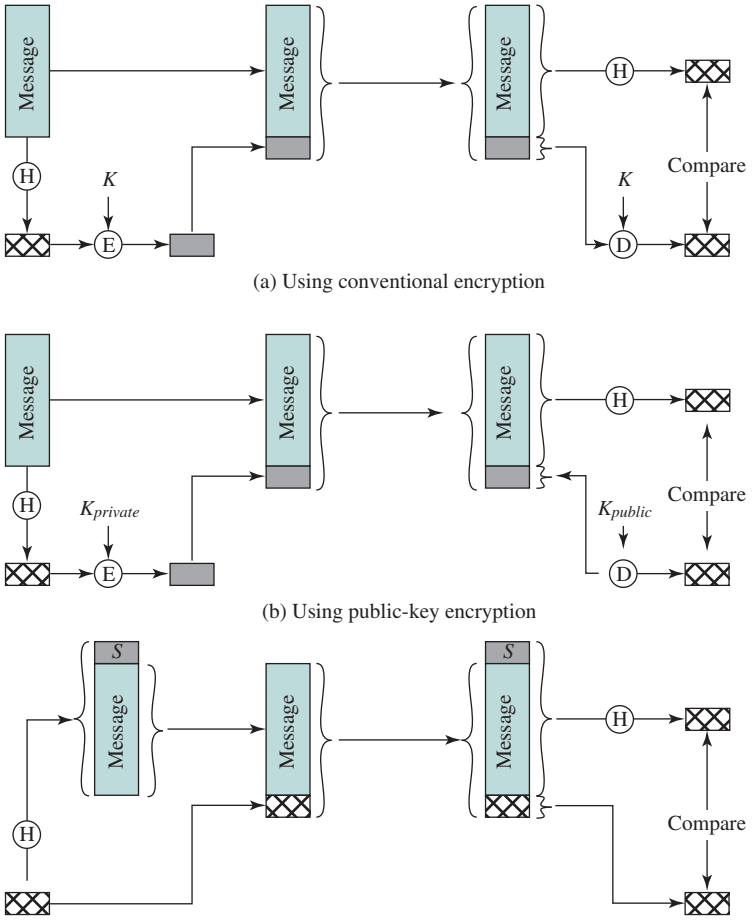


Figure K.4 Message Authentication Using a One-Way Hash Function

K.4 SECURE HASH FUNCTIONS

An essential element of many security services and applications is a secure hash function. A hash function accepts a variable-size message M as input and produces a fixed-size tag H(M), sometimes called a message digest, as output. For a digital signature, a hash code is generated for a message, encrypted with the sender's private key, and sent with the message. The receiver computes a new hash code for the incoming message, decrypts the hash code with the sender's public key and compares. If the message has been altered in transit, there will be a mismatch.

To be useful for security applications, a hash function H must have the following properties:

1. H can be applied to a block of data of any size.
2. H produces a fixed-length output.
3. H(x) is relatively easy to compute for any given x, making both hardware and software implementations practical.
4. For any given value h, it is computationally infeasible to find x such that H(x) = h. This is sometimes referred to in the literature as the **one-way property**.
5. For any given block x, it is computationally infeasible to find $y \neq x$ such that H(y) = H(x). This is sometimes referred to as **weak collision resistance**.
6. It is computationally infeasible to find any pair (x, y) such that H(x) = H(y). This is sometimes referred to as **strong collision resistance**.

In recent years, the most widely used hash function has been the Secure Hash Algorithm (SHA). SHA was developed by the NIST and published as a federal information processing standard (FIPS 180) in 1993. When weaknesses were discovered in SHA, a revised version was issued as FIPS 180-1 in 1995 and is generally referred to as SHA-1. SHA-1 produces a hash value of 160 bits. In 2002, NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512. These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1. In 2005, NIST announced the intention to phase out approval of SHA-1 and move to a reliance on the other SHA versions by 2010. Researchers have demonstrated that SHA-1 is far weaker than its 160-bit hash length suggests, necessitating the move to the newer versions of SHA.

APPENDIX L

STANDARDS ORGANIZATIONS

An important concept that recurs frequently in this book is standards. This appendix provides some background on the nature and relevance of standards and looks at the key organizations involved in developing standards for networking and communications.

L.1 THE IMPORTANCE OF STANDARDS

It has long been accepted in the telecommunications industry that standards are required to govern the physical, electrical, and procedural characteristics of communication equipment. In the past, this view has not been embraced by the computer industry. Whereas communication equipment vendors recognize that their equipment will generally interface to and communicate with other vendors' equipment, computer vendors have traditionally attempted to monopolize their customers. The proliferation of computers and distributed processing has made that an untenable position. Computers from different vendors must communicate with each other and, with the ongoing evolution of protocol standards, customers will no longer accept special-purpose protocol conversion software development. The result is that standards now permeate all the areas of technology discussed in this book.

There are a number of advantages and disadvantages to the standards-making process. The principal advantages of standards are:

- A standard assures there will be a large market for a particular piece of equipment or software. This encourages mass production and, in some cases, the use of large-scale-integration (LSI) or very-large-scale-integration (VLSI) techniques, resulting in lower costs.
- A standard allows products from multiple vendors to communicate, giving the purchaser more flexibility in equipment selection and use.

The principal disadvantages of standards are:

- A standard tends to freeze the technology. By the time a standard is developed, subjected to review and compromise, and promulgated, more efficient techniques are possible.
- There are multiple standards for the same thing. This is not a disadvantage of standards per se, but of the current way things are done. Fortunately, in recent years the various standards-making organizations have begun to cooperate more closely. Nevertheless, there are still areas where multiple conflicting standards exist.

L.2 STANDARDS AND REGULATION

It is helpful for the reader to distinguish three concepts:

- Voluntary standards
- Regulatory standards
- Regulatory use of voluntary standards

Voluntary standards are developed by standards-making organizations, such as those described in the next section. They are voluntary in that the existence of the standard does not compel its use. That is, manufacturers voluntarily implement a product that conforms to a standard if they perceive a benefit to themselves; there is no legal requirement to conform. These standards are also voluntary in the sense that they are developed by volunteers who are not paid for their efforts by the standards-making organization that administers the process. These volunteers are generally employees of interested organizations, such as manufacturers and government agencies.

Voluntary standards work because they are generally developed on the basis of broad consensus, and because the customer demand for standard products encourages the implementation of these standards by the vendors.

In contrast, a regulatory standard is developed by a government regulatory agency to meet some public objective, such as economic, health, and safety objectives. These standards have the force of regulation behind them and must be met by providers in the context in which the regulations apply. Familiar examples of regulatory standards are in areas such as fire codes and health codes. But regulations can apply to a wide variety of products, including those related to computers and communications. For example, the Federal Communications Commission regulates electromagnetic emissions.

A relatively new, or at least newly prevalent, phenomenon is the regulatory use of voluntary standards. A typical example of this is a regulation that requires that the government purchase of a product be limited to those that conform to some referenced set of voluntary standards. This approach has a number of benefits:

- It reduces the rule-making burden on government agencies.
- It encourages cooperation between government and standards organizations to produce standards of broad applicability.
- It reduces the variety of standards that providers must meet.

L.3 STANDARDS-SETTING ORGANIZATIONS

Various organizations have been involved in the development of standards related to data and computer communications. The remainder of this document provides an overview of some of the most important of these organizations:

- Internet Society
- ITU
- IEEE 802
- ISO

Internet Standards and the Internet Society

Many of the protocols that make up the TCP/IP protocol suite have been standardized or are in the process of standardization. By universal agreement, an organization known as the Internet Society is responsible for the development and publication of these standards. The Internet Society is a professional membership organization that

oversees a number of boards and task forces involved in Internet development and standardization.

This section provides a brief description of the way in which standards for the TCP/IP protocol suite are developed.

The Internet Organizations and RFC Publication The Internet Society is the coordinating committee for Internet design, engineering, and management. Areas covered include the operation of the Internet itself and the standardization of protocols used by end systems on the Internet for interoperability. Three organizations under the Internet Society are responsible for the actual work of standards development and publication:

- **Internet Architecture Board (IAB):** Responsible for defining the overall architecture of the Internet, providing guidance and broad direction to the IETF
- **Internet Engineering Task Force (IETF):** The protocol engineering and development arm of the Internet
- **Internet Engineering Steering Group (IESG):** Responsible for technical management of IETF activities and the Internet standards process

Working groups chartered by the IETF carry out the actual development of new standards and protocols for the Internet. Membership in a working group is voluntary; any interested party may participate. During the development of a specification, a working group will make a draft version of the document available as an Internet Draft, which is placed in the IETF's "Internet Drafts" online directory. The document may remain as an Internet Draft for up to six months, and interested parties may review and comment on the draft. During that time, the IESG may approve publication of the draft as an RFC (Request for Comment). If the draft has not progressed to the status of an RFC during the six-month period, it is withdrawn from the directory. The working group may subsequently publish a revised version of the draft.

The IETF is responsible for publishing the RFCs, with approval of the IESG. The RFCs are the working notes of the Internet research and development community. A document in this series may be on essentially any topic related to computer communications and may be anything from a meeting report to the specification of a standard.

The work of the IETF is divided into eight areas, each with an area director and each composed of numerous working groups. Table L.1 shows the IETF areas and their focus.

The Standardization Process The decision of which RFCs become Internet standards is made by the IESG, on the recommendation of the IETF. To become a standard, a specification must meet the following criteria:

- Be stable and well understood
- Be technically competent
- Have multiple, independent, and interoperable implementations with substantial operational experience

Table L.1 IETF Areas

IETF Area	Theme	Example Working Groups
Applications	Internet applications	Web-related protocols (HTTP) EDI-Internet integration LDAP
General	IETF processes and procedures	Policy Framework Process for Organization of Internet Standards
Internet	Internet infrastructure	IPv6 PPP extensions
Operations and management	Standards and definitions for network operations	SNMPv3 Remote Network Monitoring
Real-time applications and infrastructure	Protocols and applications for real-time requirements	Real-time Transport Protocol (RTP) Session Initiation Protocol (SIP)
Routing	Protocols and management for routing information	Multicast routing OSPF QoS routing
Security	Security protocols and technologies	Kerberos IPSec X.509 S/MIME TLS
Transport	Transport layer protocols	Differentiated services IP telephony NFS RSVP

- Enjoy significant public support
- Be recognizably useful in some or all parts of the Internet

The key difference between these criteria and those used for international standards from ITU is the emphasis here on operational experience.

The left-hand side of Figure L.1 shows the series of steps, called the *standards track*, that a specification goes through to become a standard; this process is defined in RFC 2026. The steps involve increasing amounts of scrutiny and testing. At each step, the IETF must make a recommendation for advancement of the protocol, and the IESG must ratify it. The process begins when the IESG approves the publication of an Internet Draft document as an RFC with the status of Proposed Standard.

The white boxes in the diagram represent temporary states, which should be occupied for the minimum practical time. However, a document must remain a Proposed Standard for at least six months and a Draft Standard for at least four months to allow time for review and comment. The shaded boxes represent long-term states that may be occupied for years.

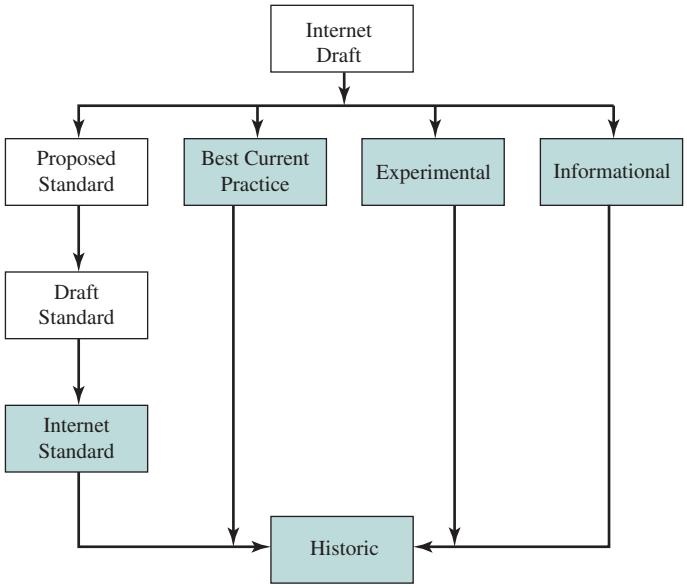


Figure L.1 Internet RFC Publication Process

For a specification to be advanced to Draft Standard status, there must be at least two independent and interoperable implementations from which adequate operational experience has been obtained.

After significant implementation and operational experience has been obtained, a specification may be elevated to Internet Standard. At this point, the Specification is assigned an STD number as well as an RFC number.

Finally, when a protocol becomes obsolete, it is assigned to the Historic state.

Internet Standards Categories All Internet standards fall into one of the two categories:

- **Technical specification (TS):** A TS defines a protocol, service, procedure, convention, or format. The bulk of the Internet standards are TSs.
- **Applicability statement (AS):** An AS specifies how, and under what circumstances, one or more TSs may be applied to support a particular Internet capability. An AS identifies one or more TSs that are relevant to the capability, and may specify values or ranges for particular parameters associated with a TS or functional subsets of a TS that are relevant for the capability.

Other RFC Types There are numerous RFCs that are not destined to become Internet standards. Some RFCs standardize the results of community deliberations about statements of principle or conclusions about what is the best way to perform some operations or IETF process function. Such RFCs are designated as Best Current Practice (BCP). Approval of BCPs follows essentially the same process for approval of Proposed Standards. Unlike standards-track documents, there is not a

three-stage process for BCPs; a BCP goes from Internet draft status to approved BCP in one step.

A protocol or other specification that is not considered ready for standardization may be published as an Experimental RFC. After further work, the specification may be resubmitted. If the specification is generally stable, has resolved known design choices, is believed to be well understood, has received significant community review, and appears to enjoy enough community interest to be considered valuable, then the RFC will be designated a Proposed Standard.

Finally, an Informational Specification is published for the general information of the Internet community.

The International Telecommunication Union

The International Telecommunication Union (ITU) is a United Nations specialized agency. Hence the members of ITU-T are governments. The U.S. representation is housed in the Department of State. The charter of the ITU is that it "is responsible for studying technical, operating, and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis." Its primary objective is to standardize, to the extent necessary, techniques and operations in telecommunications to achieve end-to-end compatibility of international telecommunication connections, regardless of the countries of origin and destination.

ITU Radio Communication Sector The ITU Radiocommunication (ITU-R) Sector was created on March 1, 1993 and comprises the former CCIR and IFRB (founded 1927 and 1947, respectively). ITU-R is responsible for all ITU's work in the field of radio communications. The main activities of ITU-R are:

- Develop draft ITU-R Recommendations on the technical characteristics of, and operational procedures for, radiocommunication services and systems.
- Compile Handbooks on spectrum management and emerging radiocommunication services and systems.

ITU-R is organized into the following study groups:

- SG 1 Spectrum management
- SG 3 Radiowave propagation
- SG 4 Fixed-satellite service
- SG 6 Broadcasting service (terrestrial and satellite)
- SG 7 Science services
- SG 8 Mobile, radiodetermination, amateur and related satellite services
- SG 9 Fixed service
- SC Special Committee on Regulatory/Procedural Matters
- CCV Coordination Committee for Vocabulary
- CPM Conference Preparatory Meeting

ITU TELECOMMUNICATION STANDARDIZATION SECTOR The ITU-T was created on March 1, 1993 as one consequence of a reform process within the ITU. It replaces the International Telegraph and Telephone Consultative Committee (CCITT), which had essentially the same charter and objectives as the new ITU-T. The ITU-T fulfills the purposes of the ITU relating to telecommunications standardization by studying technical, operating and tariff questions, and adopting Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

ITU-T is organized into 13 study groups that prepare Recommendations, numbered as follows:

1. Network and service operation
2. Tariff and accounting principles
3. Telecommunications management network and network maintenance
4. Protection against electromagnetic environment effects
5. Outside plant
6. Integrated broadband cable networks and television and sound transmission
7. Signaling requirements and protocols
8. Performance and quality of service
9. Next generation networks
10. Optical and other transport networks infrastructures
11. Multimedia terminals, systems, and applications
12. Security, languages, and telecommunication software
13. Mobile telecommunications networks

SCHEDULE Work within ITU-R and ITU-T is conducted in four-year cycles. Every four years, a World Telecommunications Standardization Conference is held. The work program for the next four years is established at the assembly in the form of questions submitted by the various study groups, based on requests made to the study groups by their members. The conference assesses the questions, reviews the scope of the study groups, creates new or abolishes existing study groups, and allocates questions to them.

Based on these questions, each study group prepares draft Recommendations. A draft Recommendation may be submitted to the next conference, four years hence, for approval. Increasingly, however, Recommendations are approved when they are ready, without having to wait for the end of the four-year study period. This accelerated procedure was adopted after the study period that ended in 1988. Thus, 1988 was the last time that a large batch of documents was published at one time as a set of Recommendations.

IEEE 802 Committee

The key to the development of the LAN market is the availability of a low-cost interface. The cost to connect equipment to a LAN must be much less than the cost of the equipment alone. This requirement, plus the complexity of the LAN logic, dictates a

solution based on the use of chips and very-large-scale integration (VLSI). However, chip manufacturers will be reluctant to commit the necessary resources unless there is a high-volume market. A widely accepted LAN standard assures volume and also enables equipment from a variety of manufacturers to intercommunicate. This is the rationale of the IEEE 802 committee.

The committee issued a set of standards, which were adopted in 1985 by the American National Standards Institute (ANSI) as American National Standards. The standards were subsequently revised and reissued as international standards by the International Organization for Standardization (ISO) in 1987, with the designation ISO 8802. Since then, the IEEE 802 committee has continued to revise and extend the standards, which are ultimately then adopted by ISO.

The committee quickly reached two conclusions. First, the task of communication across the local network is sufficiently complex that it needs to be broken up into more manageable subtasks. Accordingly, the standards are organized as a three-layer protocol hierarchy: Logical Link Control (LLC), medium access control (MAC), and physical.

Second, no single technical approach will satisfy all requirements. The second conclusion was reluctantly reached when it became apparent that no single standard would satisfy all committee participants. There was support for various topologies, access methods, and transmission media. The response of the committee was to standardize all serious proposals rather than to attempt to settle on just one. The current state of standardization is reflected by the various working groups in IEEE 802 and the work that each is doing (see Table L.2).

The International Organization for Standardization

The International Organization for Standardization, or ISO,[1] is an international agency for the development of standards on a wide range of subjects. It is a voluntary, nontreaty organization whose members are designated standards bodies of participating nations, plus nonvoting observer organizations. Although ISO is not a governmental body, more than 70% of ISO member bodies are governmental standards institutions or organizations incorporated by public law. Most of the remainder have close links with the public administrations in their own countries. The U.S. member body is the American National Standards Institute.

ISO was founded in 1946 and has issued more than 12,000 standards in a broad range of areas. Its purpose is to promote the development of standardization and related activities to facilitate international exchange of goods and services and to develop cooperation in the sphere of intellectual, scientific, technological, and economic activity. Standards have been issued to cover everything from screw threads to solar energy. One important area of standardization deals with the Open Systems Interconnection (OSI) communications architecture and the standards at each layer of the OSI architecture.

[1] ISO is not an acronym (in which case it would be IOS), but a word, derived from the Greek *isos*, meaning "equal."

Table L.2 IEEE 802 Active Working Groups

Number	Name	Charter
802.1	Higher Layer LAN Protocols	Standards and recommended practices for: 802 LAN/MAN architecture, Internet working among 802 LANs, MANs, and other wide area networks, 802 overall network management, and protocol layers above the MAC and LLC layers
802.3	Ethernet	Standards for CSMA/CD (Ethernet) based LANs
802.11	Wireless LAN	Standards for wireless LANs
802.15	Wireless Personal Area Networks	Personal area network standards for short distance wireless networks
802.16	Broadband Wireless Access	Standards for broadband wireless access
802.17	Resilient Packet Ring	Standards for RPR LAN/MAN for rates up to many gigabits per second
802.18	Radio Regulatory TAG	Monitor regulations that may affect 802.11, 802.15, and 802.16
802.19	Coexistence TAG	Standards for coexistence between wireless standards of unlicensed devices.
802.20	Mobile Broadband Wireless Access	Standards for mobile broadband wireless access
802.21	Media Independent Handoff	Standards to enable handover and interoperability between heterogeneous network types including both 802 and non-802 networks
802.22	Wireless Regional Area Networks	Standards for regional wireless networks using unused frequencies in the broadcast television band
82.23	Emergency Services	Media independent framework to provide consistent access and data that facilitate compliance to applicable civil authority requirements for communications systems that include IEEE 802 networks.

In the areas of data communications and networking, ISO standards are actually developed in a joint effort with another standards body, the International Electrotechnical Commission (IEC). IEC is primarily concerned with electrical and electronic engineering standards. In the area of information technology, the interests of the two groups overlap, with IEC emphasizing hardware and ISO focusing on software. In 1987, the two groups formed the Joint Technical Committee 1 (JTC 1). This committee has the responsibility of developing the documents that ultimately become ISO (and IEC) standards in the area of information technology.

The development of an ISO standard from first proposal to actual publication of the standard follows a six-step process. The objective is to ensure the final result is acceptable to as many countries as possible. Briefly, the steps are:

1. **Proposal stage:** A new work item is assigned to the appropriate technical committee, and within that technical committee, to the appropriate working group.
2. **Preparatory stage:** The working group prepares a working draft. Successive working drafts may be considered until the working group is satisfied that it has

developed the best technical solution to the problem being addressed. At this stage, the draft is forwarded to the working group's parent committee for the consensus-building phase.

3. **Committee stage:** As soon as a first committee draft is available, it is registered by the ISO Central Secretariat. It is distributed among interested members for balloting and technical comment. Successive committee drafts may be considered until consensus is reached on the technical content. Once consensus has been attained, the text is finalized for submission as a Draft International Standard (DIS).
4. **Enquiry stage:** The DIS is circulated to all ISO member bodies by the ISO Central Secretariat for voting and commenting within a period of five months. It is approved for submission as a Final Draft International Standard (FDIS) if a two-thirds majority is in favor and not more than one-quarter of the total number of votes cast are negative. If the approval criteria are not met, the text is returned to the originating working group for further study, and a revised document will again be circulated for voting and comment as a DIS.
5. **Approval stage:** The Final Draft International Standard (FDIS) is circulated to all ISO member bodies by the ISO Central Secretariat for a final yes/no vote within a period of two months. If technical comments are received during this period, they are no longer considered at this stage, but registered for consideration during a future revision of the International Standard. The text is approved as an International Standard if a two-thirds majority is in favor and not more than one-quarter of the total number of votes cast are negative. If these approval criteria are not met, the standard is referred back to the originating working group for reconsideration in the light of the technical reasons submitted in support of the negative votes received.
6. **Publication stage:** Once a FDIS has been approved, only minor editorial changes, if and where necessary, are introduced into the final text. The final text is sent to the ISO Central Secretariat, which publishes the International Standard.

The process of issuing an ISO standard can be a slow one. Certainly, it would be desirable to issue standards as quickly as the technical details can be worked out, but ISO must ensure the standard will receive widespread support.

APPENDIX M

SOCKETS: A PROGRAMMER'S INTRODUCTION

The concept of sockets and sockets programming was developed in the 1980s in the Unix environment as the Berkeley Sockets Interface. In essence, a socket enables communications between a client and server process and may be either connection-oriented or connectionless. A socket can be considered an endpoint in a communication. A client socket in one computer uses an address to call a server socket on another computer. Once the appropriate sockets are engaged, the two computers can exchange data.

Typically, computers with server sockets keep a TCP or UDP port open, ready for unscheduled incoming calls. The client typically determines the socket identification of the desired server by finding it in a Domain Name System (DNS) database. Once a connection is made, the server switches the dialogue to a different port number to free up the main port number for additional incoming calls.

Internet applications, such as TELNET and remote login (rlogin) make use of sockets, with the details hidden from the user. However, sockets can be constructed from within a program (in a language such as C or Java), enabling the programmer to easily support networking functions and applications. The sockets programming mechanism includes sufficient semantics to permit unrelated processes on different hosts to communicate.

The Berkeley Sockets Interface is the de facto standard application programming interface (API) for developing networking applications, spanning a wide range of operating systems. The sockets API provides generic access to interprocess communications services. Thus, the sockets capability is ideally suited for students to learn the principles of protocols and distributed applications by hands-on program development.

The Sockets Application Program Interface (API) provides a library of functions that programmers can use to develop network aware applications. It has the functionality of identifying endpoints of the connection, establishing the communication, allowing messages to be sent, waiting for incoming messages, terminating the communication, and error handling. The operating system used and the programming language both determine the specific Sockets API.

We concentrate on only two of the most widely used interfaces—the Berkley Software Distribution Sockets (BSD) as introduced for UNIX, and its slight modification the Windows Sockets (WinSock) API from Microsoft.

This sockets material is intended for the C language programmer. (It provides external references for the C++, Visual Basic, and PASCAL languages.) The Windows operating system is in the center of our discussion. At the same time, topics from the original BSD UNIX specification are introduced in order to point out (usually minor) differences in the sockets specifications for the two operating systems. Basic knowledge of the TCP/IP and UDP network protocols is assumed. Most of the code would compile on both Windows and UNIX-like systems.

We cover C language sockets exclusively, but most other programming languages, such as C++, Visual Basic, and PASCAL, can take advantage of the Winsock API, as well. The only requirement is that the language has to recognize dynamic link libraries (DLLs). In a 32-bit Windows environment, you will need to import the `wsock32.lib` to take advantage of the WinSock API. This library has to be linked, so at run time the dynamic link library `wsock32.dll` gets loaded. `wsock32.dll` runs over the TCP/IP stack. Windows NT, Windows 2000, and Windows 95 include

the file `wsock32.dll` by default. When you create your executables, if you link with `wsock32.lib` library, you will implicitly link the `wsock32.dll` at run time, without adding lines of code to your source file.

The website for this book provides links to useful Sockets websites.

M.1 SOCKETS, SOCKET DESCRIPTORS, PORTS, AND CONNECTIONS

Sockets are endpoints of communication referred to by their corresponding socket descriptors, or natural language words describing the socket's association with a particular machine or application (e.g., we will refer to a server socket as `server_s`). A connection (or socket pair) consists of the pair of IP addresses that are communicating with each other, as well a pair of port numbers, where a port number is a 32-bit positive integer usually denoted in its decimal form. Some destination port numbers are well known and indicate the type of service being connected to.

For many applications, the TCP/IP environment expects that applications use well-known ports to communicate with each other. This is done so client applications assume the corresponding server application is listening on the well-known port associated with that application. For example, the port number for HTTP, the protocol used to transfer HTML pages across the World Wide Web, is TCP port 80. By default, a Web browser will attempt to open a connection on the destination host's TCP port 80 unless another port number is specified in the URL (such as 8000 or 8080).

A *port* identifies a connection point in the local stack (i.e., port number 80 is typically used by a Web server). A *socket* identifies an IP address and port number pair (i.e., port 192.168.1.20:80 would be the Web server port 80 on host 192.168.1.20. The two together are considered a socket.). A *socket pair* identifies all four components (source address and port, and destination address and port). Since well-known ports are unique, they are sometimes used to refer to a specific application on any host that might be running the application. Using the word socket, however, would imply a specific application on some specific host. Connection, or a *socket pair*, stands for the sockets connection between two specific systems that are communicating. TCP allows multiple simultaneous connections involving the same local port number as long as the remote IP addresses or port numbers are different for each connection.

Port numbers are divided into three ranges:

- Ports 0 through 1023 are well known. They are associated with services in a static manner. For example, HTTP servers would always accept requests at port 80.
- Port numbers from 1024 through 49151 are registered. They are used for multiple purposes.
- Dynamic and private ports are those from 49152 through 65535 and services should not be associated with them.

In reality, machines start assigning dynamic ports starting at 1024. If you are developing a protocol or application that will require the use of a link, socket, port,

Proto	Local Address	Foreign Address	State
TCP	Mycomp:1025	Mycomp:0	LISTENING
TCP	Mycomp:1026	Mycomp:0	LISTENING
TCP	Mycomp:6666	Mycomp:0	LISTENING
TCP	Mycomp:6667	Mycomp:0	LISTENING
TCP	Mycomp:1234	mycomp:1234	TIME_WAIT
TCP	Mycomp:1025	2hfc327.any.com:6667	ESTABLISHED
TCP	Mycomp:1026	46c311.any.com:6668	ESTABLISHED
UDP	Mycomp:6667	*.*	

Figure M.1 Sample Netstat Output

protocol, etc., please contact the Internet Assigned Numbers Authority (IANA) to receive a port number assignment. The IANA is located at and operated by the Information Sciences Institute (ISI) of the University of Southern California. The Assigned Numbers request for comments (RFC) published by IANA is the official specification that lists port assignments. You can access it at http://www.iana.org/assignments/port-numbers.

On both UNIX and Windows, the *netstat* command can be used to check the status of all active local sockets. Figure M.1 is a sample netstat output.

M.2 THE CLIENT/SERVER MODEL OF COMMUNICATION

A socket application consists of code, executed on both communication ends. The program initiating transmission is often referred to as the client. The server, on the other hand, is a program that passively awaits incoming connections from remote clients. Server applications typically load during system startup and actively listen for incoming connections on their well-known port. Client applications will then attempt to connect to the server, and a TCP exchange will then take place. When the session is complete, usually the client will be the one to terminate the connection. Figure M.2 depicts the basic model of stream-based (or TCP/IP sockets) communication.

Running a Sockets Program on a Windows Machine Not Connected to a Network

As long as TCP/IP is installed on one machine, you can execute both the server and client code on it. (If you do not have the TCP/IP protocol stack installed, you can expect socket operations to throw exceptions such as `BindException`, `ConnectException`, `ProtocolException`, `SocketException`, etc.) You will have to use `localhost` as the hostname or `127.0.0.1` as the IP address.

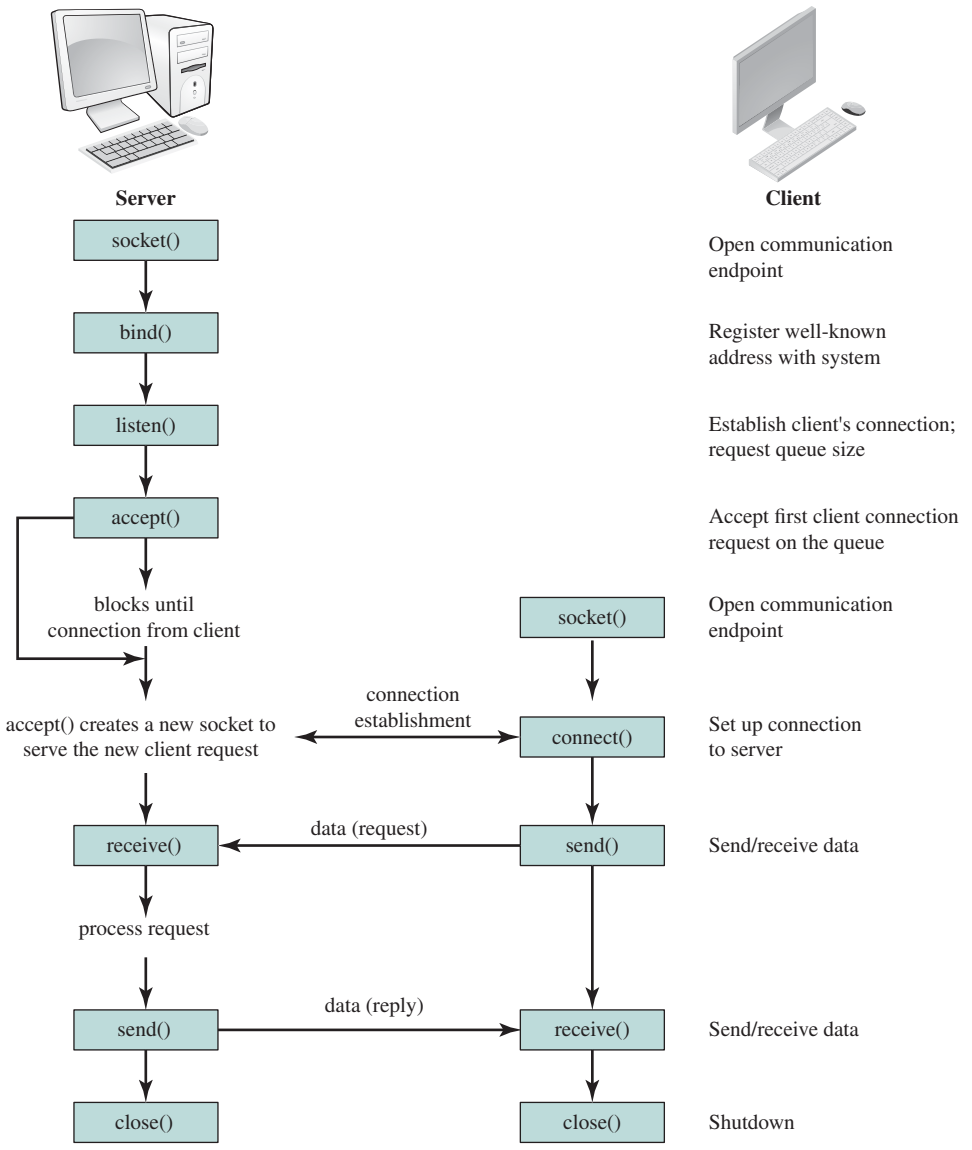


Figure M.2 Socket System Calls for Connection-Oriented Protocol

Running a Sockets Program on a Windows Machine Connected to a Network, When Both Server and Client Reside on the Same Machine

In such a case, you will be communicating with yourself. It is important to know whether your machine is attached to an Ethernet or communicates with the network through a telephone modem. In the first case, you will have an IP address assigned

to your machine, without efforts on your part. When communicating via a modem, you need to dial in, grab an IP address, and then be able to "talk to yourself." In both cases you can find out the IP address of the machine you are using with the `winipcfg` command for Win9X, and `ipconfig` for WinNT/2K and UNIX.

M.3 SOCKETS ELEMENTS

Socket Creation

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int  domain, int  type, int  protocol)
```

- *domain* is AF_UNIX, AF_INET, AF_OSI, etc. AF_INET is for communication on the Internet to IP addresses. We will only use AF_INET.
- *type* is either SOCK_STREAM (TCP, connection-oriented, reliable), or SOCK_DGRAM (UDP, datagram, unreliable), or SOCK_RAW (IP level).
- *protocol* specifies the protocol used. It is usually 0 to say we want to use the default protocol for the chosen domain and type. We always use 0.

If successful, `socket()` returns a socket descriptor, which is an integer, and −1 in the case of a failure. An example call:

```
if ((sd = socket(AF_INET, SOCK_DGRAM, 0) < 0)
  {
      printf(socket() failed.);
   exit(1);
  }
```

The Socket Address

The structures to store socket addresses as used in the domain **AF_INET**:

```
struct in_addr {
  unsigned long s_addr;
};
```

in_addr just provides a name **(s_addr)** for the C language type to be associated with IP addresses.

```
struct sockaddr_in {
 unsigned short   sin_family; // AF_INET identifiers
 unsigned short   sin_port;   // port number,
                              // if 0 then kernel chosen
struct in_addr sin_addr;      // IP address
```

```
                                // INADDR_ANY refers to the IP
                                // addresses of the current host
 char   sin_zero[8];     // Unused, always zero
};
```

Both local and remote addresses will be declared as a **sockaddr_in** structure. Depending on this declaration, **sin_addr** will represent a local or remote IP address. (On a UNIX like system, you need to include the file **<netinet/in.h>** for both structures.)

Bind to a Local Port

```
#define WIN   // WIN for Winsock and BSD for BSD sockets
#ifdef WIN
 . . .
#include <windows.h> // for all Winsock functions
 . . .
#endif
#ifdef BSD
 . . .
#include <sys/types.h>
#include <sys/socket.h> // for struct sockaddr
 . . .
#endif
int bind(int  local_s, const struct sockaddr *addr,
int  addrlen);
```

- ***local_s*** is a socket descriptor of the local socket, as created by the **socket()** function;
- ***addr*** is a pointer to the (local) address structure of this socket;
- ***addrlen*** is the length (in bytes) of the structure referenced by ***addr***.

`bind()` returns the integer 0 on success, and −1 on failure. After a call to `bind()`, a local port number is associated with the socket, but no remote destination is yet specified.

An example call:

```
struct sockaddr_in name;
...
name.sin_family = AF_INET;    // use the Internet domain
name.sin_port = htons(0);     // kernel provides a port
name.sin_addr.s_addr = htonl(INADDR_ANY); // use all IPs
                                                of host
if (bind(local_socket, (struct sockaddr *)&name,
       sizeof(name)) != 0)
                                // print error and exit
```

A call to bind is optional on the client side, but it is required on the server side. After `bind()` is called on a socket, we can retrieve its address structure, given the socket file descriptor, by using the function `getsockname()`.

Data Representation and Byte Ordering

Some computers are big endian. This refers to the representation of objects such as integers within a word. A big endian machine stores them in the expected way: the high byte of an integer is stored in the leftmost byte, while the low byte of an integer is stored in the rightmost byte. So the number $5 \times 2^{16} + 6 \times 2^8 + 4$ would be stored as:

Big endian representation		5	6	4
Little endian representation	4	6	5	
Memory (byte) address	0	1	2	3

As you can see, reading a value of the wrong word size will result in an incorrect value; when done on big endian architecture, on a little endian machine it can sometimes return the correct result. The big endian ordering is somewhat more natural to humans, because we are used to reading numbers from left to right.

A Sun SPARC is a big endian machine. When it communicates with an i-386 PC (which is a little endian), the following discrepancy will exist: The i-386 will interpret $5 \times 2^{16} + 6 \times 2^8 + 4$ as $4 \times 2^{16} + 6 \times 2^8 + 5$. To avoid this situation from occurring, the TCP/IP protocol defines a machine independent standard for byte order—network byte ordering. In a TCP/IP packet, the first transmitted data is the most significant byte. Because big endian refers to storing the most significant byte in the lowest memory address, which is the address of the data, TCP/IP defines network byte order as big endian.

Winsock uses network byte order for various values. The functions `htonl()`, `htons()`, `ntohl()`, `ntohs()` ensure the proper byte order is being used in Winsock calls, regardless of whether the computer normally uses little endian or big endian ordering.

The following functions are used to convert from host to network ordering before transmission, and from network to host form after reception:

- unsigned long *`htonl`*(unsigned long *`n`*)—host to network conversion of a 32-bit value;
- unsigned short *`htons`*(unsigned short *`n`*)—host to network conversion of a 16-bit value;
- unsigned long *`ntohl`*(unsigned long *`n`*)—network to host conversion of a 32-bit value;
- unsigned short *`ntohs`*(unsigned short *`n`*)—network to host conversion of a 16-bit value.

Connecting a Socket

A remote process is identified by an IP address and a port number. The `connect()` call evoked on the local site attempts to establish the connection to the remote destination. It is required in the case of connection-oriented communication such as

stream-based sockets (TCP/IP). Sometimes we call `connect()` on datagram sockets, as well. The reason is that this stores the destination address locally, so we do not need to specify the destination address every time when we send datagram message and thus can use the `send()` and `recv()` system calls instead of `sendto()` and `recvfrom()`. Such sockets, however, cannot be used to accept datagrams from other addresses.

```
#define WIN   // WIN for Winsock and BSD for BSD sockets
#ifdef WIN
#include <windows.h> // Needed for all Winsock functions
#endif
#ifdef BSD
#include <sys/types.h> // Needed for system defined
identifiers
#include <netinet/in.h> // Needed for Internet address
structure
#include <sys/socket.h> // Needed for socket(), bind(),
                           etc...
#endif
int  connect(int  local_s, const struct sockaddr
       *remote_addr, int  rmtaddr_len)
```

- ***local_s*** is a local socket descriptor;
- ***remote_addr*** is a pointer to protocol address of other socket;
- ***rmtaddr_len*** is the length in bytes of the address structure.

Returned is an integer 0 (on success). The Windows connect function returns a nonzero value to indicate an error, while the UNIX connection function returns a negative value in such case.

An example call:

```
#define PORT_NUM 1050          // Arbitrary port number
struct sockaddr_in serv_addr; // Server Internet address
int   rmt_s;                  // Remote socket descriptor
         // Fill-in the server (remote) socket's address
            information and connect
         // with the listening server.
server_addr.sin_family = AF_INET; // Address family to use
server_addr.sin_port = htons(PORT_NUM); // Port num to use
server_addr.sin_addr.s_addr
        = inet_addr(inet_ntoa(address));  // IP address
if (connect(rmt_s,(struct sockaddr *)&serv_addr,
        sizeof(serv_addr)) != 0)
         // print error and exit
```

The gethostbyname() Function Call

The function **gethostbyname()** is supplied a host name argument and returns NULL in case of failure, or a pointer to a **struct hostent** instance on success. It gives information about the host names, aliases, and IP addresses. This information is obtained from the DNS or a local configuration database. The **getservbyname()** will determine the port number associated with a named service. If a numeric value is supplied instead, it is converted directly to binary and used as a port number.

```
#define struct hostent {
  char  *h_name;         // official name of host
  char  **h_aliases;     // null terminated list of alias
                            names
                         // for this host
  int   h_addrtype;      // host address type,
                           e.g. AF_INET
  int   h_length;        // length of address structure
  char  **h_addr_list;   // null terminated list of
                            addresses
                         // in network byte order
};
```

Note `h_addr_list` refers to the IP address associated with the host.

```
#define WIN // WIN for Winsock and BSD for BSD sockets
#ifdef WIN
#include <windows.h> // for all Winsock functions
#endif
#ifdef BSD
#include <netdb.h> // for struct hostent
#endif
struct hostent *gethostbyname (const char *hostname);
```

Other functions that can be used to find hosts, services, protocols, or networks are: `getpeername()`, `gethostbyaddr()`, `getprotobyname()`, `getprotobynumber()`, `getprotoent()`, `getservbyname()`, `getservbyport()`, `getservent()`, `getnetbyname()`, `getnetbynumber()`, `getnetent()`.

An example call:

```
#ifdef BSD
 . . .
#include <sys\types.h> // for caddr_t type
 . . .
#endif
```

```
#define SERV_NAME somehost.somecompany.com
#define PORT_NUM 1050          // Arbitrary port number
#define h_addr h_addr_list[0]  // To hold host Internet
                                  address
 . . .
struct sockaddr_in myhost_addr; // This Internet address
struct hostent *hp;       // buffer information about
remote host
int rmt_s; // Remote socket descriptor
           // UNIX specific part
bzero( (char *)&myhost_addr, sizeof(myhost_addr) );
           // Winsock specific
memset( &myhost_addr, 0, sizeof(myhost_addr) );
           // Fill-in the server (remote) socket's
              address information and connect
           // with the listening server.
myhost_addr.sin_family = AF_INET;    // Address family
                                        to use
myhost_addr.sin_port = htons(PORT_NUM); // Port num to use
if (hp = gethostbyname(MY_NAME)== NULL)
           // print error and exit
           // UNIX specific part
bcopy(hp->h_name, (char *)&myhost_addr.sin_addr,
       hp->h_length );
           // Winsock specific
memcpy( &myhost_addr.sin_addr, hp->h_addr, hp->h_length );
if(connect(rmt_s,(struct sockaddr *)&myhost_addr,
     sizeof(myhost_addr))!=0)
           // print error and exit
```

The UNIX function `bzero()` zeroes out a buffer of specified length. It is one of a group of functions for dealing with arrays of bytes. `bcopy()` copies a specified number of bytes from a source to a target buffer. `bcmp()` compares a specified number of bytes of two byte buffers. The UNIX `bzero()` and `bcopy()` functions are not available in Winsock, so the ANSI functions `memset()` and `memcpy()` have to be used instead.

An example sockets program to get a host IP address for a given host name:

```
#define WIN   // WIN for Winsock and BSD for BSD sockets
#include <stdio.h>    // Needed for printf()
#include <stdlib.h>   // Needed for exit()
#include <string.h>   // Needed for memcpy() and strcpy()
#ifdef WIN
#include <windows.h>  // Needed for all Winsock stuff
#endif
```

```
#ifdef BSD
#include <sys/types.h>      // Needed for system defined
                               identifiers.
#include <netinet/in.h>     // Needed for Internet
                               address structure.
#include <arpa/inet.h>      // Needed for inet_ntoa.
#include <sys/socket.h>     // Needed for socket(),
                               bind(), etc...
#include <fcntl.h>
#include <netdb.h>
#endif
void main(int argc, char *argv[])
{
#ifdef WIN
WORD wVersionRequested = MAKEWORD(1,1);
                            // Stuff for WSA functions
WSADATA wsaData;            // Stuff for WSA functions
#endif
struct hostent *host;   // Structure for gethostbyname()
struct in_addr address; // Structure for Internet
address
char host_name[256];    // String for host name
if (argc != 2)
{
printf(*** ERROR - incorrect number of command line
arguments \n);
printf( usage is 'getaddr host_name' \n);
exit(1);
}
#ifdef WIN
                        // Initialize winsock
WSAStartup(wVersionRequested, &wsaData);
#endif
                        // Copy host name into host_name
strcpy(host_name, argv[1]);
                        // Do a gethostbyname()
printf(Looking for IP address for '%s'... \n,
       host_name);
host = gethostbyname(host_name);
                        // Output address if host found
if (host == NULL)
printf( IP address for '%s' could not be found \n,
        host_name);
else
```

```
{
memcpy(&address, host->h_addr, 4);
printf( IP address for '%s' is %s \n, host_name,
        inet_ntoa(address));
}
#ifdef WIN
                                  // Cleanup winsock
WSACleanup();
#endif
}
```

Listening for an Incoming Client Connection

The `listen()` function is used on the server in the case of connection-oriented communication to prepare a socket to accept messages from clients. It has the prototype:

```
int listen(int  sd, int  qlen);
```

- ***sd*** is a socket descriptor of a socket after a **`bind()`** call
- ***qlen*** specifies the maximum number of incoming connection requests that can wait to be processed by the server while the server is busy.

The call to **`listen()`** returns an integer: 0 on success, and −1 on failure. For example:

```
if (listen(sd, 5) < 0) {
       // print error and exit
```

Accepting a Connection from a Client

The `accept()` function is used on the server in the case of connection-oriented communication (after a call to `listen()`) to accept a connection request from a client.

```
#define WIN   // WIN for Winsock and BSD for BSD sockets
#ifdef WIN
 . . .
#include <windows.h>        // for all Winsock functions
 . . .
#endif
#ifdef BSD
 . . .
#include <sys/types.h>
#include <sys/socket.h>     // for struct sockaddr
 . . .
#endif
int accept(int  server_s, struct sockaddr *  client_addr,
int *  clntaddr_len)
```

- ***server_s*** is a socket descriptor the server is listening on
- ***client_addr*** will be filled with the client address
- ***clntaddr_len*** contains the length of the client address structure.

The `accept()` function returns an integer representing a new socket (−1 in case of failure).

Once executed, the first queued incoming connection is accepted, and a new socket with the same properties as *sd* is created and returned. It is the socket that the server will use from now on to communicate with this client. Multiple successful calls to `connect()` will result in multiple new sockets returned.

An example call:

```
struct sockaddr_in client_addr;
int server_s, client_s, clntaddr_len;
 ...
if ((client_s = accept(server_s, (struct sockaddr *)&
client_addr, &clntaddr_len) < 0)
                // print error and exit
                // at this stage a thread or a process
                   can take over and handle
               // communication with the client
```

Successive calls to accept on the same listening socket return different connected sockets. These connected sockets are multiplexed on the same port of the server by the running TCP stack functions.

Sending and Receiving Messages on a Socket

We will present only four function calls in this section. There are, however, more than four ways to send and receive data through sockets. Typical functions for TCP/IP sockets are **`send()`** and **`recv()`**.

```
int send(int  socket, const void  *msg, unsigned
     int  msg_length, int  flags);
int recv(int  socket, void  *rcv_buff, unsigned
     int buff_length, int  flags);
```

- ***socket*** is the local socket used to send and receive.
- ***msg*** is the pointer to a message.
- ***msg_length*** is the message length.
- ***rcv_buff*** is a pointer to the receive buffer.
- ***buff_length*** is its length.
- ***flags*** changes the default behavior of the call.

For example, a particular value of flags will be used to specify that the message is to be sent without using local routing tables (they are used by default).

Typical functions for UDP sockets are:

```
int sendto(int  socket, const void  *msg, unsigned
     int  msg_length, int  flags, struct sockaddr
     *dest_addr, unsigned int addr_length);
int recvfrom(int  socket, void  *rcv_buff, unsigned
     int buff_length, int  flags, struct sockaddr
     *src_addr, unsigned int addr_length);
```

Most parameters are the same as for **`send()`** and **`recv()`**, except ***`dest_addr/src_addr`*** and ***`addr_length`***. Unlike with stream sockets, datagram callers of **`sendto()`** need to be informed of the destination address to send the message to, and callers of **`recvfrom()`** need to distinguish between different sources sending datagram messages to the caller. We provide code for TCP/IP and UDP client and server applications in the following sections, where you can find the sample calls of all four functions.

Closing a Socket

The prototype:

```
int closesocket(int  sd); // Windows prototype
int close(int  fd); // BSD UNIX prototype
```

fd and *sd* are a file descriptor (same as socket descriptor in UNIX) and a socket descriptor.

When a socket on some reliable protocol, such as TCP/IP is closed, the kernel will still retry to send any outstanding data, and the connection enters a `TIME_WAIT` state (see Figure M.1). If an application picks the same port number to connect to, the following situation can occur. When this remote application calls `connect()`, the local application assumes that the existing connection is still active and sees the incoming connection as an attempt to duplicate an existing connection. As a result, `[WSA]ECONNREFUSED` error is returned. The operating system keeps a reference counter for each active socket. A call to `close()` is essentially decrementing this counter on the argument socket. This is important to keep in mind when we are using the same socket in multiple processes. We will provide a couple of example calls in the code segments presented in the next subsections.

Report errors

All the preceding operations on sockets can exhibit a number of different failures at execution time. It is considered a good programming practice to report the returned error. Most of these errors are designed to assist the developer in the debugging process, and some of them can be displayed to the user, as well. In a Windows environment, all of the returned errors are defined in `winsock.h`. On an UNIX-like system, you can find these definitions in `socket.h`. The Windows codes are

computed by adding 10,000 to the original BSD error number and adding the prefix `WSA` in front of the BSD error name. For example:

Windows name	BSD name	Windows value	BSD value
`WSAEPROTOTYPE`	`EPROTOTYPE`	10041	41

There are a few Windows-specific errors not present in a UNIX system:

`WSASYSNOTREADY`	*10091*	Returned by `WSAStartup()` indicating that the network subsystem is unusable.
`WSAVERNOTSUPPORTED`	*10092*	Returned by `WSAStartup()` indicating that the Windows Sockets DLL cannot support this app.
`WSANOTINITIALISED`	*10093*	Returned by any function except `WSAStartup()`, when a successful `WSAStartup()` has not yet been performed.

An example error-catching source file, responsible for displaying an error and exiting:

```
#ifdef WIN
#include <stdio.h>      // for fprintf()
#include <winsock.h>    // for WSAGetLastError()
#include <stdlib.h>     // for exit()
#endif
#ifdef BSD
#include <stdio.h>      // for fprintf() and perror()
#include <stdlib.h>     // for exit()
#endif

void catch_error(char * program_msg)
{
char err_descr[128];    // to hold error description
int err;
err = WSAGetLastError();
                        // record the winsock.h error
                           description
if (err == WSANO_DATA)
   strcpy(err_descr, WSANO_DATA (11004) Valid name, no
"       data record of requested type.);
if (err == WSANO_RECOVERY)
   strcpy(err_descr, WSANO_RECOVERY (11003) This is a
      non-recoverable error.);
if (err == WSATRY_AGAIN)
```

```
 . . .
fprintf(stderr,%s: %s\n, program_msg, err_descr);
exit(1);
}
```

You can extend the list of errors to be used in your Winsock application by looking at http://www.sockets.com.

Example TCP/IP Client Program (Initiating Connection)

This client program is designed to receive a single message from a server (lines 39–41) then terminate itself (lines 45–56). It sends a confirmation to the server after the message is received (lines 42–44).

```
#define WIN   // WIN for Winsock and BSD for BSD sockets
#include      // Needed for printf()
#include      // Needed for memcpy() and strcpy()
#ifdef WIN
#include      // Needed for all Winsock stuff
#endif
#ifdef BSD
#include      // Needed for system defined identifiers.
#include      // Needed for Internet address structure.

#include      // Needed for socket(), bind(), etc...
#include      // Needed for inet_ntoa()
#include
#include
#endif
#define PORT_NUM 1050  // Port number used at the server
#define IP_ADDR 131.247.167.101  // IP address of server
                                    (*** HARDWIRED ***)
void main(void)
{
#ifdef WIN
WORD wVersionRequested = MAKEWORD(1,1); // WSA functions
WSADATA wsaData;       // WSA functions
#endif
unsigned int server_s; // Server socket descriptor
struct sockaddr_in server_addr; // Server Internet address
char out_buf[100];     // 100-byte output buffer for data
char in_buf[100];      // 100-byte input buffer for data
#ifdef WIN             // Initialize Winsock
WSAStartup(wVersionRequested, &wsaData);
#endif
```

```
      // Create a socket
server_s = socket(AF_INET, SOCK_STREAM, 0);
      // Fill-in the server socket's address and do a
         connect with
      // the listening server. The connect() will block.
Server_addr.sin_family = AF_INET;      // Address family
Server_addr.sin_port = htons(PORT_NUM); // Port num
Server_addr.sin_addr.s_addr = inet_addr(IP_ADDR);
                           // IP address
Connect(server_s, (struct sockaddr *)&server_addr,
sizeof(server_addr));
                           // Receive from the server
recv(server_s, in_buf, sizeof(in_buf), 0);
printf(Received from server... data = '%s' \n, in_buf);
                           // Send to the server
strcpy(out_buf, Message -- client to server);
send(server_s, out_buf, (strlen(out_buf) + 1), 0);
                           // Close all open sockets
#ifdef WIN
closesocket(server_s);
#endif
#ifdef BSD
close(server_s);
#endif
#ifdef WIN                // Cleanup winsock
WSACleanup();
#endif
}
```

Example TCP/IP Server Program (Passively Awaiting Connection)

All the following server program does is serving a message to a client running on another host. It creates one socket in line 37 and listens for a single incoming service request from the client through this single socket. When the request is satisfied, this server terminates (lines 62–74).

```
#define WIN // WIN for Winsock and BSD for BSD sockets
#include <stdio.h> // Needed for printf()
#include <string.h> // Needed for memcpy() and strcpy()
#ifdef WIN
#include <windows.h> // Needed for all Winsock calls
#endif
#ifdef BSD
```

```
#include <sys/types.h>   // Needed for system defined
                            identifiers.
#include <netinet/in.h>  // Needed for Internet address
structure.
#include <sys/socket.h>  // Needed for socket(), bind(),
                             etc...
#include <arpa/inet.h>   // Needed for inet_ntoa()
#include <fcntl.h>
#include <netdb.h>
#endif
#define PORT_NUM 1050    // Arbitrary port number for
                            the server
#define MAX_LISTEN 3     // Maximum number of listens
                            to queue
void main(void)
{
#ifdef WIN
WORD wVersionRequested = MAKEWORD(1,1);
                          // for WSA functions
WSADATA wsaData;         // for WSA functions
#endif
unsigned int server_s;   // Server socket descriptor
struct sockaddr_in server_addr;
                          // Server Internet address
unsigned int client_s;  // Client socket descriptor
struct sockaddr_in client_addr;
                          // Client Internet address
struct in_addr client_ip_addr; // Client IP address
int addr_len;             // Internet address length
char out_buf[100];       // 100-byte output buffer for data
char in_buf[100];        // 100-byte input buffer for data
#ifdef WIN               // Initialize Winsock
WSAStartup(wVersionRequested, &wsaData);
#endif                  // Create a socket
                        // - AF_INET is Address Family
                     Internet and SOCK_STREAM is streams
server_s = socket(AF_INET, SOCK_STREAM, 0);
                         // Fill-in my socket's address
                           information and bind the socket
                         // - See winsock.h for a
                            description of struct
                            sockaddr_in
server_addr.sin_family = AF_INET; // Address family
                                     to use
```

```
server_addr.sin_port = htons(PORT_NUM);
                // Port number to use
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
                // Listen on any IP addr.
bind(server_s, (struct sockaddr *)&server_addr,
sizeof(server_addr));
  // Listen for connections (queueing up to MAX_LISTEN)
listen(server_s, MAX_LISTEN);
  // Accept a connection. The accept() will block and
     then return with
  // client_addr filled-in.
addr_len = sizeof(client_addr);
client_s = accept(server_s, (struct sockaddr *)&client_
addr, &addr_len);
  // Copy the four-byte client IP address into an IP
     address structure
  // - See winsock.h for a description of struct in_addr
memcpy(&client_ip_addr, &client_addr.sin_addr.s_addr, 4);
  // Print an informational message that accept completed
printf(Accept completed!!! IP address of client = %s
port = %d \n,
inet_ntoa(client_ip_addr), ntohs(client_addr.sin_port));
                // Send to the client
strcpy(out_buf, Message -- server to client);
send(client_s, out_buf, (strlen(out_buf) + 1), 0);
                // Receive from the client
recv(client_s, in_buf, sizeof(in_buf), 0);
printf(Received from client... data = '%s' \n, in_buf);
                // Close all open sockets
#ifdef WIN
closesocket(server_s);
closesocket(client_s);
#endif
#ifdef BSD
close(server_s);
close(client_s);
#endif
#ifdef WIN
                // Cleanup Winsock
WSACleanup();
#endif
}
```

This is not a very realistic implementation. More often, server applications will contain some indefinite loop and be able to accept multiple requests. The preceding code can be easily converted into such more realistic server by inserting lines 46–61 into a loop in which the termination condition is never satisfied (e.g., **`while(1){ . . . }`**). Such servers will create one permanent socket through the **`socket()`** call (line 37), while a temporary socket gets spun off every time when a request is accepted (line 49). In this manner, each temporary socket will be responsible of handling a single incoming connection. If a server gets killed eventually, the permanent socket will be closed, as will each of the active temporary sockets. The TCP implementation determines when the same port number will become available for reuse by other applications. The status of such port will be in TIME-WAIT state for some predetermined period of time, as shown in Figure M.1 for port number 1234.

M.4 STREAM AND DATAGRAM SOCKETS

When sockets are used to send a connection-oriented, reliable stream of bytes across machines, they are of SOCK_STREAM type. As we previously discussed, in such cases sockets have to be connected before being used. The data are transmitted through a bidirectional stream of bytes and are guaranteed to arrive in the order they were sent.

Sockets of SOCK_DGRAM type (or datagram sockets) support a bidirectional flow of data, as well, but data may arrive out of order, and possibly duplicated (i.e., it is not guaranteed to be arriving in sequence or to be unique). Datagram sockets also do not provide reliable service since they can fail to arrive at all. It is important to note, though, that the data record boundaries are preserved, as long as the records are no longer than the receiver could handle. Unlike stream sockets, datagram sockets are connectionless; hence, they do not need to be connected before being used. Figure M.3 shows the basic flowchart of datagram sockets communication. Taking the stream-based communication model as a base, as one can easily notice, the calls to `listen()` and `accept()` are dropped, and the calls to `send()` and `recv()` are replaced by calls to `sendto()` and `recvfrom()`.

Example UDP Client Program (Initiate Connections)

```
#define WIN   // WIN for Winsock and BSD for BSD sockets
#include <stdio.h>     // Needed for printf()
#include <string.h>    // Needed for memcpy() and strcpy()
#ifdef WIN
#include <windows.h> // Needed for all Winsock stuff
#endif
#ifdef BSD
```

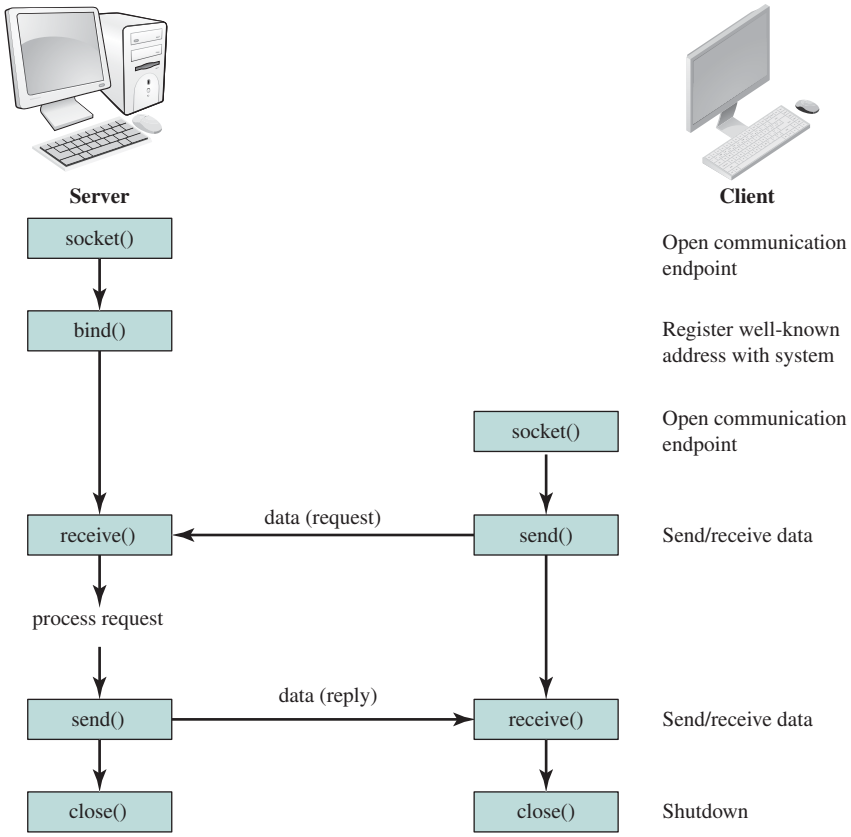


Figure M.3 Socket System Calls for Connectionless Protocol

```
#include <sys/types.h>   // Needed for system defined
                            identifiers.
#include <netinet/in.h> // Needed for Internet address
structure.
#include <sys/socket.h> // Needed for socket(), bind(),
                           etc...
#include <arpa/inet.h>  // Needed for inet_ntoa()
#include <fcntl.h>
#include <netdb.h>
#endif
#define PORT_NUM 1050    // Port number used
#define IP_ADDR 131.247.167.101
           // IP address of server1 (*** HARDWIRED ***)
void main(void)
{
#ifdef WIN
```

```
WORD wVersionRequested = MAKEWORD(1,1);
                         // Stuff for WSA functions
WSADATA wsaData;          // Stuff for WSA functions
#endif
unsigned int server_s;   // Server socket descriptor
struct sockaddr_in server_addr;
                         // Server Internet address
int addr_len;             // Internet address length
char out_buf[100];     // 100-byte buffer for output data
char in_buf[100];      // 100-byte buffer for input data
#ifdef WIN             // This stuff initializes winsock
WSAStartup(wVersionRequested, &wsaData);
#endif                 // Create a socket
                       // - AF_INET is Address Family
                          Internet and SOCK_DGRAM is
                          datagram
server_s = socket(AF_INET, SOCK_DGRAM, 0);
                       // Fill-in server1 socket's
                          address information
server_addr.sin_family = AF_INET;
                      // Address family to use
 server_addr.sin_port = htons(PORT_NUM);
                       // Port num to use
 server_addr.sin_addr.s_addr = inet_addr(IP_ADDR);
                       // IP address to use
                    // Assign a message to buffer out_buf
 strcpy(out_buf, Message from client1 to server1);
                    // Now send the message to server1.
                       The + 1 includes the end-of-string
                    // delimiter
 sendto(server_s, out_buf, (strlen(out_buf) + 1), 0,
 (struct sockaddr *)&server_addr, sizeof(server_addr));
                    // Wait to receive a message
 addr_len = sizeof(server_addr);
 recvfrom(server_s, in_buf, sizeof(in_buf), 0,
 (struct sockaddr *)&server_addr, &addr_len);
                    // Output the received message
 printf(Message received is: '%s' \n, in_buf);
                    // Close all open sockets
#ifdef WIN
 closesocket(server_s);
#endif
#ifdef BSD
 close(server_s);
```

```
#endif
#ifdef WIN
 // Cleanup Winsock
 WSACleanup();
#endif
}
```

Example UDP Server Program (Passively Await Connection)

```
#define WIN   // WIN for Winsock and BSD for BSD sockets
#include <stdio.h>     // Needed for printf()
#include <string.h>    // Needed for memcpy() and strcpy()
#ifdef WIN
#include <windows.h>  // Needed for all Winsock stuff
#endif
#ifdef BSD
#include <sys/types.h>     // Needed for system defined
                              identifiers.
#include <netinet/in.h>    // Needed for Internet address
                              structure.
#include <sys/socket.h>    // Needed for socket(),
                              bind(), etc...
#include <arpa/inet.h>     // Needed for inet_ntoa()
#include <fcntl.h>
#include <netdb.h>
#endif
#define PORT_NUM 1050      // Port number used
#define IP_ADDR 131.247.167.101 // IP address of client1
void main(void)
{
#ifdef WIN
WORD wVersionRequested = MAKEWORD(1,1);
                           // Stuff for WSA functions
WSADATA wsaData;           // Stuff for WSA functions
#endif
unsigned int server_s;     // Server socket descriptor
struct sockaddr_in server_addr;
                           // Server1 Internet address
struct sockaddr_in client_addr; // Client1 Internet
address
int addr_len;              // Internet address length
char out_buf[100];     // 100-byte buffer for output data
char in_buf[100];      // 100-byte buffer for input data
long int i;            // Loop counter
```

```
#ifdef WIN    // This stuff initializes winsock
WSAStartup(wVersionRequested, &wsaData);
#endif        // Create a socket
              // AF_INET is Address Family Internet and
                 SOCK_DGRAM is datagram
server_s = socket(AF_INET, SOCK_DGRAM, 0);
              // Fill-in my socket's address information
server_addr.sin_family = AF_INET;     // Address family
server_addr.sin_port = htons(PORT_NUM);   // Port number
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
              // Listen on any IP address
bind(server_s, (struct sockaddr *)&server_addr,
sizeof(server_addr));
        // Fill-in client1 socket's address information
client_addr.sin_family = AF_INET;
        // Address family to use
client_addr.sin_port = htons(PORT_NUM); // Port num to use
client_addr.sin_addr.s_addr = inet_addr(IP_ADDR);
        // IP address to use
        // Wait to receive a message from client1
addr_len = sizeof(client_addr);
recvfrom(server_s, in_buf, sizeof(in_buf), 0,
(struct sockaddr *)&client_addr, &addr_len);
        // Output the received message
printf(Message received is: '%s' \n, in_buf);
        // Spin-loop to give client1 time to turn-around
for (i=0; i>> Step #5 <<<
        // Now send the message to client1. The + 1
           includes the end-of-string
        // delimiter
sendto(server_s, out_buf, (strlen(out_buf) + 1), 0,
(struct sockaddr *)&client_addr, sizeof(client_addr));
        // Close all open sockets
#ifdef WIN
closesocket(server_s);
#endif
#ifdef BSD
close(server_s);
#endif
#ifdef WIN
        // Cleanup Winsock
WSACleanup();
#endif
}
```

M.5 RUN-TIME PROGRAM CONTROL

Nonblocking Socket Calls

By default, a socket is created as blocking, (i.e., it blocks until the current function call is completed). For example, if we execute an `accept()` on a socket, the process will block until there is an incoming connection from a client. In UNIX, two functions are involved in turning a blocking socket into a nonblocking one: `ioctl()` and `select()`. The first facilitates input/output control on a file descriptor or socket. The `select()` function is then used to determine the socket status—ready or not ready to perform action.

```
// change the blocking state of a socket
unsigned long unblock = TRUE;
            // TRUE for nonblocking, FALSE for blocking
ioctl(s, FIONBIO, &unblock);
```

We then call accept() periodically:

```
while(client_s = accept(s, NULL, NULL) > 0)
{
 if ( client_s == EWOULDBLOCK)
      // wait until a client connection arrives,
      // while executing useful tasks
 else
      // process accepted connection
}     // display error and exit
```

or use the `select()` function call to query the socket status, as in the following segment from a nonblocking socket program:

```
if (select(max_descr + 1, &sockSet, NULL, NULL,
&sel_timeout) == 0)
              // print a message for the user
else
{ . . .
client_s = accept(s, NULL, NULL);
  . . .
}
```

In this way, when some socket descriptor is ready for I/O, the process has to be constantly polling the OS with `select()` calls, until the socket is ready. Although the process executing a `select()` call would suspend the program until the socket is ready or until the `select()` function times out (as opposed to suspending it until the socket is ready, if the socket were blocking), this solution is still inefficient. Just like calling a nonblocking `accept()` within a loop, calling `select()` within a loop results in wasting CPU cycles.

Asynchronous I/O (Signal Driven I/O)

A better solution is to use asynchronous I/O (i.e., when I/O activity is detected on the socket, the OS informs the process immediately and thus relieves it from the burden of polling all the time). In the original BSD UNIX, this involves the use of calls to `sigaction()` and `fcntl()`. An alternative to poll for the status of a socket through the `select()` call is to let the kernel inform the application about events via a `SIGIO` signal. In order to do that, a valid signal handler for `SIGIO` must be installed with `sigaction()`. The following program does not involve sockets, it merely provides a simple example on how to install a signal handler. It catches an interrupt char (Cntrl-C) input by setting the signal handling for `SIGINT` (interrupt signal) via `sigaction()`:

```
#include <stdio.h>          // for printf()
#include <sys/signal.h>     // for sigaction()
#include <unistd.h>         // for pause()
void catch_error(char *errorMessage);
                            // for error handling
void InterruptSignalHandler(int signalType);
                            // handle interr. signal
int main(int argc, char *argv[])
{
 struct sigaction handler;
                           // Signal handler specification
   // Set InterruptSignalHandler() as a handler function
 handler.sa_handler = InterruptSignalHandler;
                            // Create mask for all signals
 if (sigfillset(&handler.sa_mask) < 0)
 catch_error(sigfillset() failed);

                            // No flags
 handler.sa_flags = 0;
            // Set signal handling for interrupt signals
 if (sigaction(SIGINT, &handler, 0) < 0)
 catch_error(sigaction() failed);
 for(;;)
 pause();   // suspend program until signal received
 exit(0);
}
void InterruptSignalHandler(int signalType)
{
 printf(Interrupt Received. Program terminated.\n);
 exit(1);
}
```

A **FASYNC** flag must be set on a socket file descriptor via **fcntl()**. In more detail, first we notify the OS about our desire to install a new disposition for **SIGIO**, using **sigaction()**; then we force the OS to submit signals to the current process by using **fcntl()**. This call is needed to ensure that among all processes that access the socket, the signal is delivered to the current process (or process group); next, we use **fcntl()** again to set the status flag on the same socket descriptor for asynchronous **FASYNC**. The following segment of a datagram sockets program follows this scheme. Note all the unnecessary details are omitted for clarity:

```
int main()
{
 . . .
        // Create socket for sending/receiving datagrams
        // Set up the server address structure
        // Bind to the local address
        // Set signal handler for SIGIO
        // Create mask that mask all signals
 if (sigfillset(&handler.sa_mask) < 0)
        // print error and exit
        // No flags
 handler.sa_flags = 0;
 if (sigaction(SIGIO, &handler, 0) < 0)
        // print error and exit
        // We must own the socket to receive the SIGIO
           message
 if (fcntl(s_socket, F_SETOWN, getpid()) < 0)
        //print error and exit
        // Arrange for asynchronous I/O and SIGIO
           delivery
 if (fcntl(s_socket, F_SETFL, FASYNC | O_NONBLOCK) < 0)
        // print error and exit
 for (;;)
 pause();
 . . .
}
```

Under Windows, the `select()` function is not implemented. The `WSAAsyncSelect()` is used to request notification of network events (i.e., request that `Ws2_32.dll` sends a message to the window `hWnd`):

```
WSAAsyncSelect (SOCKET  socket, HWND  hWnd,
unsigned int  wMsg, long  lEvent)
```

The SOCKET type is defined in `winsock.h`. *socket* is a socket descriptor, *hWnd* is the window handle, *wMsg* is the message, *lEvent* is usually a logical OR of all events we are expecting to be notified of, when completed. Some of the event values

are FD_CONNECT (connection completed), FD_ACCEPT (ready to accept), FD_READ (ready to read), FD_WRITE (ready to write), FD_CLOSE (connection closed). You can easily incorporate the following stream sockets program segment into the previously presented programs or your own application. (Again, details are omitted.):

```
        // the message for the asynchronous notification
 #define wMsg (WM_USER + 4)
 ...
       // socket_s has already been created and bound to
          a name
       // listen for connections
if (listen(socket_s, 3) == SOCKET_ERROR)
       // print error message
       // exit after cleanup
       // get notification on connection accept
       // to this window
if (WSAAsyncSelect(s, hWnd, wMsg, FD_ACCEPT)==
SOCKET_ERROR)
       // print cannot process asynchronously
       // exit after cleanup
 else  // accept the incoming connection
```

Further references on Asynchronous I/O are "The Pocket Guide to TCP/IP Sockets—C version" by Donahoo and Calvert (for UNIX), and "Windows Sockets Network Programming" (for Windows), by Bob Quinn.

M.6 REMOTE EXECUTION OF A WINDOWS CONSOLE APPLICATION

Simple sockets operations can be used to accomplish tasks that are otherwise hard to achieve. For example, by using sockets we can remotely execute an application. The sample code[1] is presented. Two sockets programs, a local and remote, are used to transfer a Windows console application (an `.exe` file) from the local host to the remote host. The program is executed on the remote host, then `stdout` is returned to the local host.

Local Code

```
#include <stdio.h>   // Needed for printf()
#include <stdlib.h>  // Needed for exit()
#include <string.h>  // Needed for memcpy() and strcpy()
#include <windows.h> // Needed for Sleep() and Winsock
                        stuff
```

[1] This and other code presented is in part written by Ken Christensen and Karl S. Lataxes at the Computer Science Department of the University of South Florida, http://www.csee.usf.edu/~christen/tools/.

```
#include <fcntl.h>     // Needed for file i/o constants
#include <sys\stat.h>  // Needed for file i/o constants
#include <io.h> // Needed for open(), close(), and eof()
#define PORT_NUM 1050  // Arbitrary port number for the
                          server
#define MAX_LISTEN 1   // Maximum number of listens to
                          queue
#define SIZE 256       // Size in bytes of transfer
                          buffer

void main(int argc, char *argv[])
{
WORD wVersionRequested = MAKEWORD(1,1); // WSA functions
WSADATA wsaData;       // Winsock API data structure
unsigned int remote_s; // Remote socket descriptor
struct sockaddr_in remote_addr;
                       // Remote Internet address
struct sockaddr_in server_addr;
                       // Server Internet address
unsigned char bin_buf[SIZE]; // Buffer for file transfer
unsigned int fh;       // File handle
unsigned int length;   // Length of buffers transferred
struct hostent *host;  // Structure for gethostbyname()
struct in_addr address; // Structure for Internet
                           address
char host_name[256];   // String for host name
int addr_len; // Internet address length
unsigned int local_s;  // Local socket descriptor
struct sockaddr_in local_addr; // Local Internet address
struct in_addr remote_ip_addr; // Remote IP address
   // Check if number of command line arguments is valid
if (argc !=4)
{
printf( *** ERROR - Must be 'local (host) (exefile)
        (outfile)' \n);
printf( where host is the hostname *or* IP address \n);
printf( of the host running remote.c, exefile is the \n);
printf( name of the file to be remotely run, and \n);
printf( outfile is the name of the local output file. \n);
exit(1);
}
        // Initialization of winsock
WSAStartup(wVersionRequested, &wsaData);
        // Copy host name into host_name
```

```
strcpy(host_name, argv[1]);
        // Do a gethostbyname()
host = gethostbyname(argv[1]);
if (host == NULL)
{
printf( *** ERROR - IP address for '%s' not be found \n,
host_name);
exit(1);
}         // Copy the four-byte client IP address into
             an IP address structure
memcpy(&address, host->h_addr, 4);
        // Create a socket for remote
remote_s = socket(AF_INET, SOCK_STREAM, 0);
        // Fill-in the server (remote) socket's address
information and connect
        // with the listening server.
server_addr.sin_family = AF_INET; // Address family to use
server_addr.sin_port = htons(PORT_NUM); // Port num to use
server_addr.sin_addr.s_addr = inet_addr(inet_
ntoa(address)); // IP address
connect(remote_s, (struct sockaddr *)&server_addr,
sizeof(server_addr));
        // Open and read *.exe file
if((fh = open(argv[2], O_RDONLY | O_BINARY, S_IREAD |
     S_IWRITE)) == -1)
{
printf( ERROR - Unable to open file '%s'\n, argv[2]);
exit(1);
}
        // Output message stating sending executable file
printf(Sending '%s' to remote server on '%s' \n,
argv[2], argv[1]);
        // Send *.exe file to remote
while(!eof(fh))
{
length = read(fh, bin_buf, SIZE);
send(remote_s, bin_buf, length, 0);
}
        // Close the *.exe file that was sent to the
           server (remote)
close(fh);
        // Close the socket
closesocket(remote_s);
        // Cleanup Winsock
```

```
WSACleanup();
        // Output message stating remote is executing
printf( '%s' is executing on remote server \n, argv[2]);
        // Delay to allow everything to cleanup
Sleep(100);
        // Initialization of winsock
WSAStartup(wVersionRequested, &wsaData);

        // Create a new socket to receive output file
           from remote server
local_s = socket(AF_INET, SOCK_STREAM, 0);
        // Fill-in the socket's address information and
           bind the socket
local_addr.sin_family = AF_INET; // Address family to use
local_addr.sin_port = htons(PORT_NUM); // Port num to use
local_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        // Listen on any IP addr
bind(local_s, (struct sockaddr *)&local_addr,
sizeof(local_addr));
        // Listen for connections (queueing up to
           MAX_LISTEN)
listen(local_s, MAX_LISTEN);
        // Accept a connection, the accept will block
           and then return with
        // remote_addr filled in.
addr_len = sizeof(remote_addr);
remote_s = accept(local_s, (struct sockaddr*)
       &remote_addr, &addr_len);
        // Copy the four-byte client IP address into an
           IP address structure
memcpy(&remote_ip_addr, &remote_addr.sin_addr.s_addr, 4);
        // Create and open the output file for writing
if ((fh=open(argv[3], O_WRONLY | O_CREAT | O_TRUNC |
       O_BINARY,
S_IREAD | S_IWRITE)) == - 1)
{
printf( *** ERROR - Unable to open '%s'\n, argv[3]);
exit(1);
}
        // Receive output file from server
length = SIZE;
while(length > 0)
{
length = recv(remote_s, bin_buf, SIZE, 0);
```

```
write(fh, bin_buf, length);
}
  // Close output file that was received from the remote
close(fh);
  // Close the sockets
closesocket(local_s);
closesocket(remote_s);
  // Output final status message
printf(Execution of '%s' and transfer of output
     to '%s' done! \n,
argv[2], argv[3]);
  // Cleanup Winsock
WSACleanup();
}
```

Remote Code

```
#include <stdio.h>     // Needed for printf()
#include <stdlib.h>    // Needed for exit()

#include <string.h>    // Needed for memcpy() and strcpy()
#include <windows.h>  // Needed for Sleep() and Winsock
                         stuff
#include <fcntl.h>     // Needed for file i/o constants
#include <sys\stat.h> // Needed for file i/o constants
#include <io.h>  // Needed for open(), close(), and eof()
#define PORT_NUM 1050 // Arbitrary port number for the
                         server
#define MAX_LISTEN 1   // Maximum number of listens to
                         queue
#define IN_FILE run.exe // Name given to transferred
                           *.exe file
#define TEXT_FILE output
                       // Name of output file for stdout
#define SIZE 256       // Size in bytes of transfer
                         buffer
void main(void)
{
WORD wVersionRequested = MAKEWORD(1,1); // WSA functions
WSADATA wsaData;        // WSA functions
unsigned int remote_s; // Remote socket descriptor
struct sockaddr_in remote_addr;
                       // Remote Internet address
struct sockaddr_in server_addr;
                       // Server Internet address
```

```
unsigned int local_s;          // Local socket descriptor
struct sockaddr_in local_addr; // Local Internet address
struct in_addr local_ip_addr;  // Local IP address
int addr_len;                  // Internet address length
unsigned char bin_buf[SIZE]; // File transfer buffer
unsigned int fh;               // File handle
unsigned int length;          // Length of transf. buffers
                               // Do forever
while(1)
{                              // Winsock initialization
WSAStartup(wVersionRequested, &wsaData);
                              // Create a socket
remote_s = socket(AF_INET, SOCK_STREAM, 0);
             // Fill-in my socket's address information
                and bind the socket
remote_addr.sin_family = AF_INET; // Address family to use
remote_addr.sin_port = htons(PORT_NUM);
             // Port number to use
remote_addr.sin_addr.s_addr = htonl(INADDR_ANY);
             // Listen on any IP addr
bind(remote_s,  (struct  sockaddr  *)&remote_addr,
sizeof(remote_addr));
             // Output waiting message
printf(Waiting for a connection... \n);
   // Listen for connections (queueing up to MAX_LISTEN)
listen(remote_s, MAX_LISTEN);
   // Accept a connection, accept() will block and return
      with local_addr
addr_len = sizeof(local_addr);
local_s = accept(remote_s, (struct sockaddr *)&local_
addr, &addr_len);
   // Copy the four-byte client IP address into an IP
      address structure
memcpy(&local_ip_addr, &local_addr.sin_addr.s_addr, 4);
   // Output message acknowledging receipt, saving of *.exe
printf( Connection established, receiving remote executable
file \n);
   // Open IN_FILE for remote executable file
if((fh = open(IN_FILE, O_WRONLY | O_CREAT | O_TRUNC |
O_BINARY,
S_IREAD | S_IWRITE)) == - 1)
{
printf( *** ERROR - unable to open executable file \n);
exit(1);
```

```
}
     // Receive executable file from local
length = 256;
while(length > 0)
{
length = recv(local_s, bin_buf, SIZE, 0);
write(fh, bin_buf, length);
}
     // Close the received IN_FILE
close(fh);
     // Close sockets
closesocket(remote_s);
closesocket(local_s);
     // Cleanup Winsock
WSACleanup();
     // Print message acknowledging execution of *.exe
printf( Executing remote executable (stdout to output
file) \n);
     // Execute remote executable file (in IN_FILE)
system(IN_FILE > TEXT_FILE);
     // Winsock initialization to reopen socket to send
        output file to local
WSAStartup(wVersionRequested, &wsaData);
     // Create a socket
     // - AF_INET is Address Family Internet and SOCK_
        STREAM is streams
local_s = socket(AF_INET, SOCK_STREAM, 0);
     // Fill in the server's socket address information
        and connect with
     // the listening local
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT_NUM);
server_addr.sin_addr.s_addr = inet_addr(inet_ntoa
        (local_ip_addr));
connect(local_s, (struct sockaddr *)&server_addr,
sizeof(server_addr));
     // Print message acknowledging transfer of output to
        client
printf( Sending output file to local host \n);
     // Open output file to send to client
if((fh = open(TEXT_FILE, O_RDONLY |
        O_BINARY, S_IREAD | S_IWRITE)) == - 1)
{
printf( *** ERROR - unable to open output file \n);
```

```
exit(1);
}
                // Send output file to client
while(!eof(fh))
{
length = read(fh, bin_buf, SIZE);
send(local_s, bin_buf, length, 0);
}
                // Close output file
close(fh);
                // Close sockets
closesocket(remote_s);
closesocket(local_s);
                // Cleanup Winsock
WSACleanup();
                // Delay to allow everything to cleanup
Sleep(100);
}
}
```

APPENDIX N

THE INTERNATIONAL REFERENCE ALPHABET

A familiar example of data is **text** or character strings. While textual data are most convenient for human beings, they cannot, in character form, be easily stored or transmitted by data processing and communications systems. Such systems are designed for binary data. Thus, a number of codes have been devised by which characters are represented by a sequence of bits. Perhaps the earliest common example of this is the Morse code. Today, the most commonly used text code is the International Reference Alphabet (IRA).[1] Each character in this code is represented by a unique 7-bit binary code; thus, 128 different characters can be represented. Table N.1 lists all of the code values. In the table, the bits of each character are labeled from b_7, which is the most significant bit, to b_1, the least significant bit. Characters are of two types: printable and control (see Table N.2). Printable characters are the alphabetic, numeric, and special characters that can be printed on paper or displayed on a screen. For example, the bit representation of the character "K" is $b_7b_6b_5b_4b_3b_2b_1 = 1001011$. Some of the control characters have to do with controlling the printing or displaying

Table N.1 The International Reference Alphabet (IRA)

Bit Position											
	b_7			0	0	0	0	1	1	1	1
		b_6		0	0	1	1	0	0	1	1
			b_5	0	1	0	1	0	1	0	1
b_4	b_3	b_2	b_1								
0	0	0	0	NUL	DLE	SP	0	@	P	‘	p
0	0	0	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	STX	DC2	”	2	B	R	b	r
0	0	1	1	ETX	DC3	#	3	C	S	c	s
0	1	0	0	EOT	DC4	$	4	D	T	d	t
0	1	0	1	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	ACK	SYN	&	6	F	V	f	v
0	1	1	1	BEL	ETB	’	7	G	W	g	w
1	0	0	0	BS	CAN	(	8	H	X	h	x
1	0	0	1	HT	EM	)	9	I	Y	i	y
1	0	1	0	LF	SUB	*	:	J	Z	j	z
1	0	1	1	VT	ESC	+	;	K	[	k	{
1	1	0	0	FF	FS	,	<	L	\	l	|
1	1	0	1	CR	GS	–	=	M	]	m	}
1	1	1	0	SO	RS	.	>	N	^]	n	~
1	1	1	1	SI	US	/	?	O	_	o	DEL

[1] IRA is defined in ITU-T Recommendation T.50 and was formerly known as International Alphabet Number 5 (IA5). The U.S. national version of IRA is referred to as the American Standard Code for Information Interchange (ASCII).

Table N.2 IRA Control Characters

Format Control	
BS (Backspace): Indicates movement of the printing mechanism or display cursor backward one position. **HT** (Horizontal Tab): Indicates movement of the printing mechanism or display cursor forward to the next preassigned "tab" or stopping position. **LF** (Line Feed): Indicates movement of the printing mechanism or display cursor to the start of the next line.	**VT** (Vertical Tab): Indicates movement of the printing mechanism or display cursor to the next of a series or preassigned printing lines. **FF** (Form Feed): Indicates movement of the printing mechanism or display cursor to the starting position of the next page, form, or screen. **CR** (Carriage Return): Indicates movement of the printing mechanism or display cursor to the starting position of the same line.
Transmission Control	
SOH (Start of Heading): Used to indicate the start of a heading, which may contain address or routing information. **STX** (Start of Text): Used to indicate the start of the text and so also indicates the end of the heading. **ETX** (End of Text): Used to terminate the text that was started with STX. **EOT** (End of Transmission): Indicates the end of a transmission, which may have included one or more "texts" with their headings. **ENQ** (Enquiry): A request for a response from a remote station. It may be used as a "WHO ARE YOU" request for a station to identify itself.	**ACK** (Acknowledge): A character transmitted by a receiving device as an affirmation response to a sender. It is used as a positive response to polling messages. **NAK** (Negative Acknowledgment): A character transmitted by a receiving device as a negative response to a sender. It is used as a negative response to polling messages. **SYN** (Synchronous/Idle): Used by a synchronous transmission system to achieve synchronization. When no data is being sent a synchronous transmission system may send SYN characters continuously. **ETB** (End of Transmission Block): Indicates the end of a block of data for communication purposes. It is used for blocking data where the block structure is not necessarily related to the processing format.
Information Separator	
FS (File Separator) **GS** (Group Separator) **RS** (Record Separator) **US** (Unit Separator)	Information separators to be used in an optional manner except that their hierarchy shall be FS (the most inclusive) to US (the least inclusive)
Miscellaneous	
NUL (Null): No character. Used for filling in time or filling space on tape when there are no data. **BEL** (Bell): Used when there is need to call human attention. It may control alarm or attention devices. **SO** (Shift Out): Indicates the code combinations that follow shall be interpreted as outside of the standard character set until a SI character is reached. **SI** (Shift In): Indicates the code combinations that follow shall be interpreted according to the standard character set. **DEL** (Delete): Used to obliterate unwanted characters; for example by overwriting. **SP** (Space): A nonprinting character used to separate words, or to move the printing mechanism or display cursor forward by one position.	**DLE** (Data Link Escape): A character that shall change the meaning of one or more contiguously following characters. It can provide supplementary controls, or permits the sending of data characters having any bit combination. **DC1, DC2, DC3, DC4** (Device Controls): Characters for the control of ancillary devices or special terminal features. **CAN** (Cancel): Indicates the data that precedes it in a message or block should be disregarded (usually because an error has been detected). **EM** (End of Medium): Indicates the physical end of a tape or other medium, or the end of the required or used portion of the medium. **SUB** (Substitute): Substituted for a character that is found to be erroneous or invalid. **ESC** (Escape): A character intended to provide code extension in that it gives a specified number of continuously following characters an alternate meaning.

of characters; an example is carriage return. Other control characters are concerned with communications procedures.

IRA-encoded characters are almost always stored and transmitted using 8 bits per character. In that case, the eighth bit is a parity bit used for error detection. The parity bit is the most significant bit and is therefore labeled b_8. This bit is set such that the total number of binary 1s in each octet is always odd (odd parity) or always even (even parity). Thus a transmission error that changes a single bit, or any odd number of bits, can be detected.

APPENDIX O

BACI: THE BEN-ARI CONCURRENT PROGRAMMING SYSTEM

O.1 INTRODUCTION

In Chapter 5, concurrency concepts are introduced (e.g., mutual exclusion and the critical section problem) and synchronization techniques are proposed (e.g., semaphores, monitors, and message passing). Deadlock and starvation issues for concurrent programs are discussed in Chapter 6. Due to the increasing emphasis on parallel and distributed computing, understanding concurrency and synchronization is more necessary than ever. To obtain a thorough understanding of these concepts, practical experience writing concurrent programs is needed.

Three options exist for this desired "hands-on" experience. First, we can write concurrent programs with an established concurrent programming language such as Concurrent Pascal, Modula, Ada, or the SR Programming Language. To experiment with a variety of synchronization techniques, however, we must learn the syntax of many concurrent programming languages. Second, we can write concurrent programs using system calls in an operating system such as UNIX. It is easy, however, to be distracted from the goal of understanding concurrent programming by the details and peculiarities of a particular operating system (e.g., details of the semaphore system calls in UNIX). Lastly, we can write concurrent programs with a language developed specifically for giving experience with concurrency concepts such as the Ben-Ari Concurrent Interpreter (BACI). Using such a language offers a variety of synchronization techniques with a syntax that is usually familiar. Languages developed specifically for giving experience with concurrency concepts are the best option to obtain the desired hands-on experience.

Section O.2 contains a brief overview of the BACI system and how to obtain the system. Section O.3 contains examples of BACI programs, and Section O.4 contains a discussion of projects for practical concurrency experience at the implementation and programming levels. Lastly, Section O.5 contains a description of enhancements to the BACI system that have been made.

O.2 BACI

System Overview

BACI is a direct descendant of Ben-Ari's modification to sequential Pascal (Pascal-S). Pascal-S is a subset of standard Pascal by Wirth, without files, except INPUT and OUTPUT, sets, pointer variables, and goto statements. Ben-Ari took the Pascal-S language and added concurrent programming constructs such as the `cobegin... coend` construct and the semaphore variable type with `wait` and `signal` operations. BACI is Ben-Ari's modification to Pascal-S with additional synchronization features (e.g., monitors) as well as encapsulation mechanisms to ensure that a user is prevented from modifying a variable inappropriately (e.g., a semaphore variable should only be modified by semaphore functions).

BACI simulates concurrent process execution and supports the following synchronization techniques: general semaphores, binary semaphores, and monitors. The BACI system is composed of two subsystems, as illustrated in Figure O.1. The first

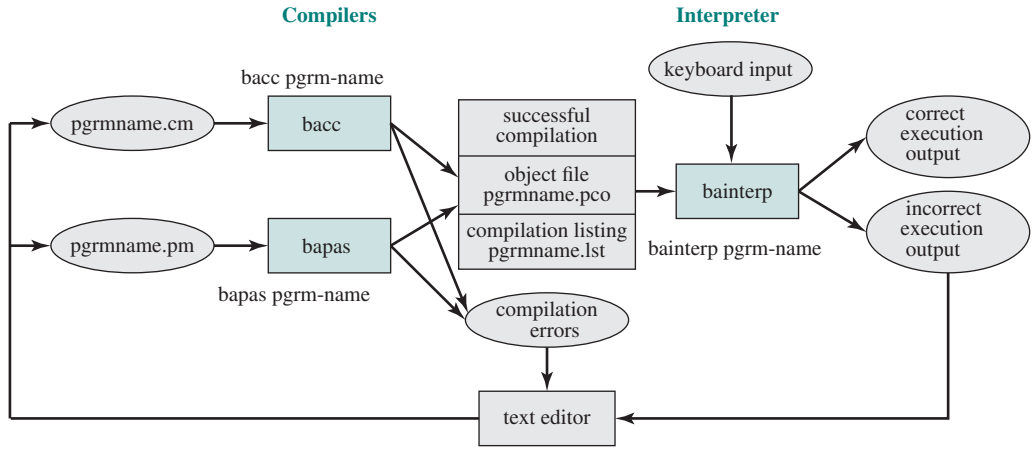


Figure O.1

subsystem, the compiler, compiles a user's program into intermediate object code, called PCODE. There are two compilers available with the BACI system, corresponding to two popular languages taught in introductory programming courses. The syntax of one compiler is similar to standard Pascal; BACI programs that use the Pascal syntax are denoted as pgrm-name.pm. The syntax of the other compiler is similar to standard C++; these BACI programs are denoted as pgrm-name.cm. Both compilers create two files during the compilation: pgrm-name.lst and pgrm-name.pco.

The second subsystem in the BACI system, the interpreter, executes the object code created by the compiler. In other words, the interpreter executes pgrm-name.pco. The core of the interpreter is a preemptive scheduler; during execution, this scheduler randomly swaps between concurrent processes, thus simulating a parallel execution of the concurrent processes. The interpreter offers a number of different debug options, such as single-step execution, disassembly of PCODE instructions, and display of program storage locations.

Concurrency Constructs in BACI

In the rest of this appendix, we focus on the compiler similar to standard C++. We call this compiler C−−; although the syntax is similar to C++, it does not include inheritance, encapsulation, or other object-oriented programming features. In this section, we give an overview of the BACI concurrency constructs; see the user's guides at the BACI website for further details of the required Pascal or C−− BACI syntax.

COBEGIN A list of processes to be run concurrently is enclosed in a `cobegin` block. Such blocks cannot be nested and must appear in the main program.

```
cobegin { proc1(...); proc2(...); ... ; procN(...); }
```

The PCODE statements created by the compiler for the above block are interleaved by the interpreter in an arbitrary, "random" order; multiple executions of the same program containing a `cobegin` block will appear to be nondeterministic.

SEMAPHORES A semaphore in BACI is a nonnegative-valued `int` variable, which can only be accessed by the semaphore calls defined subsequently. A binary semaphore in BACI, one that only assumes the values 0 and 1, is supported by the `binarysem` subtype of the `semaphore` type. During compilation and execution, the compiler and interpreter enforce the restrictions that a `binarysem` variable can only have the values 0 or 1 and that `semaphore` type can only be nonnegative. BACI semaphore calls include

- `initialsem(semaphore sem, int expression)`
- `p(semaphore sem)`: If the value of `sem` is greater than zero, then the interpreter decrements `sem` by one and returns, allowing p's caller to continue. If the value of `sem` is equal to zero, then the interpreter puts `p`'s caller to sleep. The command `wait` is accepted as a synonym for `p`.
- `v(semaphore sem)`: If the value of `sem` is equal to zero and one or more processes are sleeping on `sem`, then wake up one of these processes. If no processes are waiting on `sem`, then increment `sem` by one. In any event, `v`'s caller is allowed to continue. (BACI conforms to Dijkstra's original semaphore proposal by randomly choosing which process to wake up when a signal arrives.) The command `signal` is accepted as a synonym for `v`.

MONITORS BACI supports the monitor concept, with some restrictions. A monitor is a C++ block, like a block defined by a procedure or function, with some additional properties (e.g., conditional variables). In BACI, a monitor must be declared at the outermost, global level and it cannot be nested with another monitor block. Three constructs are used by the procedures and functions of a monitor to control concurrency: condition variables, `waitc` (wait on a condition), and `signalc` (signal a condition). A condition never actually has a value; it is somewhere to wait or something to signal. A monitor process can wait for a condition to hold or signal that a given condition now holds through the `waitc` and `signalc` calls. `waitc` and `signalc` calls have the following syntax and semantics:

- `waitc(condition cond, int prio)`: The monitor process (and hence the outside process calling the monitor process) is blocked on the condition `cond` and assigned the priority `prio`.
- `waitc(condition cond)`: This call has the same semantics as the `waitc` call, but the `wait` is assigned a default priority of 10.
- `signalc(condition cond)`: Wake some process waiting on `cond` with the smallest (highest) priority; if no process is waiting on `cond`, do nothing.

BACI conforms to the immediate resumption requirement. In other words, a process waiting on a condition has priority over a process trying to enter the monitor, if the process waiting on a condition has been signaled.

OTHER CONCURRENCY CONSTRUCTS The C−− BACI compiler provides several low-level concurrency constructs that can be used to create new concurrency control primitives. If a function is defined as atomic, then the function is nonpreemptible. In other words, the interpreter will not interrupt an atomic function with a context switch. In BACI, the suspend function puts the calling process to sleep and the revive function revives a suspended process.

How to Obtain BACI

The BACI system, with two user guides (one for each of the two compilers) and detailed project descriptions, is available at the BACI website at http://inside.mines.edu/fs_home/tcamp/baci/baci_index.html. The BACI system is written in both C and Java. The C version of the BACI system can be compiled in Linux, RS/6000 AIX, Sun OS, DOS, and CYGWIN on Windows with minimal modifications to the Makefile file. (See the README file in the distribution for installation details for a given platform.)

O.3 EXAMPLES OF BACI PROGRAMS

In Chapters 5 and 6, a number of the classical synchronization problems were discussed (e.g., the readers/writers problem and the dining philosophers problem). In this section, we illustrate the BACI system with three BACI programs. Our first example illustrates the nondeterminism in the execution of concurrent processes in the BACI system. Consider the following program:

```
const int m = 5;
int n;
void incr(char id)
{
 int i;
     for(i = 1; i <= m; i = i + 1)
      {
        n = n + 1;
        cout << id << " n =" << n << " i =";
        cout << i << " " << id << endl;
      }
}
main()
{
     n = 0;
     cobegin {
```

```
        incr( 'A'); incr( 'B' ); incr('C');
        }
      cout << "The sum is " << n << endl;
}
```

Note in the preceding program that if each of the three processes created (A, B, and C) executed sequentially, the output sum would be 15. Concurrent execution of the statement n = n + 1; however, can lead to different values of the output sum. After we compiled the preceding program with BACI, we executed the PCODE file with `bainterp` a number of times. Each execution produced output sums between 9 and 15. One sample execution produced by the BACI interpreter is the following.

```
Source file: incremen.cm Fri Aug 1 16:51:00 1997
CB n = 2 i =1 C n =2
A n = 2 i =1 i =1 A
CB
     n = 3 i = 2 C
A n = 4 i = 2 C n = 5 i = 3 C
A
B n = 6C i = 2 B
     n = 7 i = 4 C
A n = 8 i = 3 A
BC n = 10 n = 10 i = 5 C
A n = i = 311 i = 4 A
 B
A n = 12 i = B5 n = 13A
     i = 4 B
B n = 14 i = 5 B
The sum is 14
```

Special machine instructions are needed to synchronize the access of processes to a common main memory. Mutual exclusion protocols, or synchronization primitives, are then built on top of these special instructions. In BACI, the interpreter will not interrupt a function defined as atomic with a context switch. This feature allows users to implement these low-level special machine instructions. For example, the following program is a BACI implementation of the testset function. A testset instruction tests the value of the function's argument i. If the value of i is zero, the function replaces it with 1 and returns true; otherwise, the function does not change the value of i and returns false. As discussed in Section 5.2, special machine instructions (e.g., testset) allow more than one action to occur without interruption. BACI has an atomic keyword defined for this purpose.

```
// Test and set instruction
//
atomic int testset(int& i)
```

```
{
     if (i == 0) {
      i = 1;
      return 1;
     }
     else
      return 0;
}
```

We can use testset to implement mutual exclusion protocols, as shown in the following program. This program is a BACI implementation of a mutual exclusion program based on the test and set instruction. The program assumes three concurrent processes; each process requests mutual exclusion 10 times.

```
int bolt = 0;
const int RepeatCount = 10;
void proc(int id)
{
     int i = 0;
     while(i < RepeatCount) {
      while (testset(bolt)); // wait
            // enter critical section
      cout << id;
            // leave critical section
      bolt = 0;
      i++;
     }
}
main()
{
     cobegin {
      proc(0); proc(1); proc(2);
     }
}
```

The following two programs are a BACI solution to the bounded-buffer producer/consumer problem with semaphores (see Figure 5.13). In this example, we have two producers, three consumers, and a buffer size of five. We first list the program details for this problem. We then list the included file that defines the bounded-buffer implementation.

```
// A solution to the bounded-buffer producer/consumer
problem
// Stallings,  Figure 5.13
// bring in the bounded-buffer machinery
```

```
#include "boundedbuff.inc"
const int ValueRange = 20; // integers in 0..19 will be
                                produced
semaphore to; // for exclusive access to terminal output
semaphore s; // mutual exclusion for the buffer
semaphore n; // # consumable items in the buffer
semaphore e; // # empty spaces in the buffer
int produce(char id)
{
     int tmp;
     tmp = random(ValueRange);
     wait(to);
     cout << "Producer " << id << " produces " << tmp
       << endl;
     signal(to);
     return tmp;
}
void consume(char id, int i)
{
     wait(to);
     cout << "Consumer " << id << " consumes " << i
       << endl;
     signal(to);
}
void producer(char id)
{
     int i;
     for (;;) {
      i = produce(id);
      wait(e);
      wait(s);
      append(i);
      signal(s);
      signal(n);
     }
}
void consumer(char id)
{
     int i;
     for (;;) {
      wait(n);
      wait(s);
      i = take();
      signal(s);
```

```
        signal(e);
        consume(id,i);
      }
}
main()
{
      initialsem(s,1);
      initialsem(n,0);
      initialsem(e,SizeOfBuffer);
      initialsem(to,1);
      cobegin {
       producer('A'); producer('B');
       consumer('x'); consumer('y'); consumer('z');
      }
}
```

```
// boundedbuff.inc -- bounded buffer include file
const int SizeOfBuffer = 5;
int buffer[SizeOfBuffer];
int in = 0; // index of buffer to use for next append
int out = 0; // index of buffer to use for next take
void append(int v)
      // add v to the buffer
      // overrun is assumed to be taken care of
      // externally through semaphores or conditions
{
      buffer[in] = v;
      in = (in + 1) % SizeOfBuffer;
}
int take()
      // return an item from the buffer
 // underrun is assumed to be taken care of
 // externally through a semaphore or condition
{
      int tmp;
      tmp = buffer[out];
      out = (out + 1) % SizeOfBuffer;
      return tmp;
}
```

One sample execution of the preceding bounded-buffer solution in BACI is the following.

```
Source file: semprodcons.cm Fri Aug 1 12:36:55 1997
Producer B produces 4
```

```
Producer A produces 13
Producer B produces 12
Producer A produces 4
Producer B produces 17
Consumer x consumes 4
Consumer y consumes 13
Producer A produces 16
Producer B produces 11
Consumer z consumes 12
Consumer x consumes 4
Consumer y consumes 17
Producer B produces 6
...
```

O.4 BACI PROJECTS

In this section, we discuss two general types of projects one can implement in BACI. We first discuss projects that involve the implementation of low-level operations (e.g., special machine instructions that are used to synchronize the access of processes to a common main memory). We then discuss projects that are built on top of these low-level operations (e.g., classical synchronization problems). For more information on these projects, see the project descriptions included in the BACI distribution. For solutions to some of these projects, teachers should contact the authors. In addition to the projects discussed in this section, many of the problems at the end of Chapter 5 and Appendix A can be implemented in BACI.

Implementation of Synchronization Primitives

Implementation of Machine Instructions There are numerous machine instructions that one can implement in BACI. For example, one can implement the compare-and-swap or the exchange instruction discussed in Figure 5.2. The implementation of these instructions should be based on an atomic function that returns an int value. You can test your implementation of the machine instruction by building a mutual exclusion protocol on top of your low-level operation.

Implementation of Fair Semaphores (FIFO) The semaphore operation in BACI is implemented with a random wake up order, which is how semaphores were originally defined by Dijkstra. As discussed in Section 5.3, however, the fairest policy is FIFO. One can implement semaphores with this FIFO wake up order in BACI. At least the following four procedures should be defined in the implementation:

- `CreateSemaphores()` to initialize the program code
- `InitSemaphore(int sem-index)` to initialize the semaphore represented by `sem-index`
- `FIFOP(int sem-index)`
- `FIFOV(int sem-index)`

This code needs to be written as a system implementation and, as such, should handle all possible errors. In other words, the semaphore designer is responsible for producing code that is robust in the presence of ignorant, stupid, or even malicious use by the user community.

Semaphores, Monitors, and Implementations

There are many classical concurrent programming problems: the producer/consumer problem, the dining philosophers, the reader/writer problem with different priorities, the sleeping barber problem, and the cigarette smoker's problem. All of these problems can be implemented in BACI. In this section, we discuss nonstandard semaphore/monitor projects that one can implement in BACI to further aid the understanding of concurrency and synchronization concepts.

A'S AND B'S AND SEMAPHORES For the following program outline in BACI,

```
// global semaphore declarations here
void A()
{
     p()'s and v()'s ONLY
}
void B()
{
     p()'s and v()'s ONLY
}
main()
{
     // semaphore initialization here
     cobegin {
      A(); A(); A(); B(); B();
     }
}
```

complete the program using the least number of general semaphores, such that the processes ALWAYS terminate in the order A (any copy), B (any copy), A (any copy), A, B. Use the -t option of the interpreter to display process termination. (Many variations of this project exist. For example, have four concurrent processes terminated in the order ABAA or eight concurrent processes terminated in the order AABABABB.)

USING BINARY SEMAPHORES Repeat the previous project using binary semaphores. Evaluate why assignment and IF-THEN-ELSE statements are necessary in this solution, although they were not necessary in solutions to the previous project. In other words, explain why you cannot use only Ps and Vs in this case.

BUSY WAITING VERSUS SEMAPHORES Compare the performance of a solution to mutual exclusion that uses busy waiting (e.g., the testset instruction) to a solution that uses semaphores. For example, compare a semaphore solution and a testset solution

to the ABAAB project discussed previously. In each case, use a large number of executions (say, 1000) to obtain better statistics. Discuss your results, explaining why one implementation is preferred over another.

SEMAPHORES AND MONITORS In the spirit of Problem 5.17, implement a monitor using general semaphores, then implement a general semaphore using a monitor in BACI.

GENERAL AND BINARY SEMAPHORES Prove that general semaphores and binary semaphores are equally powerful, by implementing one type of semaphore with the other type of semaphore and vice versa.

TIME TICKS: A MONITOR PROJECT Write a program containing a monitor AlarmClock. The monitor must have an `int` variable `theClock` (initialized to zero) and two functions:

- `Tick()`: This function increments `theClock` each time that it is called. It can do other things, like `signalc`, if needed.
- `int Alarm(int id, int delta)`: This function blocks the caller with identifier `id` for at least `delta` ticks of `theClock`.

The main program should have two functions as well:

- `void Ticker()`: This procedure calls `Tick()` in a repeat-forever loop.
- `void Thread(int id, int myDelta)`: This function calls `Alarm` in a repeat-forever loop.

You may endow the monitor with any other variables that it needs. The monitor should be able to accommodate up to five simultaneous alarms.

A PROBLEM OF A POPULAR BAKER Due to the recent popularity of a bakery, almost every customer needs to wait for service. To maintain service, the baker wants to install a ticket system that will ensure customers are served in turn. Construct a BACI implementation of this ticket system.

O.5 ENHANCEMENTS TO THE BACI SYSTEM

We have enhanced the BACI System in several ways:

1. We have implemented the BACI system in Java (JavaBACI). This, along with our original C implementation of BACI, is available from: http://inside.mines.edu/fs_home/tcamp/baci/baci_index.html. The JavaBACI classes and source files are stored in self-extracting Java .jar files. JavaBACI includes all BACI applications: C and Pascal compilers, disassembler, archiver, linker, and command-line and GUI PCODE interpreters. The input, behavior, and output of programs in JavaBACI are identical to the input, behavior, and output of programs in our C implementation of BACI; we note that, in JavaBACI, students

continue to write concurrency programs in C−− or Pascal (not Java). JavaBACI will execute on any computer that has an installation of the Java Virtual Machine.

2. We have added graphical user interfaces (GUIs) for JavaBACI and the UNIX version of BACI in C. The windowing environments of these GUIs allow a user to monitor all aspects of the execution of a BACI program; specifically, a user can set and remove breakpoints (either by PCODE address or source line), view variables values, runtime stacks, and process tables, and examine interleaved PCODE execution. The BACI GUIs are available at the BACI GUI Web site http://inside.mines.edu/fs_home/tcamp/baci/index_gui.html. For an alternative GUI, see below.
3. We have created a distributed version of BACI. Similar to concurrent programs, it is difficult to prove the correctness of distributed programs without an implementation. Distributed BACI allows distributed programs to be easily implemented. In addition to proving the correctness of a distributed program, one can use distributed BACI to test the program's performance. Distributed BACI is available at the following website: http://inside.mines.edu/fs_home/tcamp/baci/dbaci.html.
4. We have a PCODE disassembler that will provide the user with an annotated listing of a PCODE file, showing the mnemonics for each PCODE instruction and, if available, the corresponding program source that generated the instruction. This PCODE disassembler is included in the BACI System.
5. We have added the capability of separate compilation and external variables to both compilers (C and Pascal). The BACI System includes an archiver and a linker that enable the creation and use of libraries of BACI PCODE. For more details, see the BACI Separate Compilation User's Guide.

The BACI system has also been enhanced by others.

1. David Strite, an M.S. student who worked with Linda Null from the Pennsylvania State University, created a BACI Debugger: A GUI Debugger for the BACI System. This GUI is available at http://cs.hbg.psu.edu/~null/baci.
2. Using BACI and the BACI GUI from Pennsylvania State University, Moti Ben-Ari from the Weizmann Institute of Science in Israel created an integrated development environment for learning concurrent programming by simulating concurrency called jBACI. jBACI is available at: https://code.google.com/archive/p/jbaci/.

APPENDIX P

PROCEDURE CONTROL

A common technique for controlling the execution of procedure calls and returns makes use of a stack. This appendix summarizes the basic properties of stacks and looks at their use in procedure control.

P.1 STACK IMPLEMENTATION

A stack is an ordered set of elements, only one of which (the most recently added) can be accessed at a time. The point of access is called the *top* of the stack. The number of elements in the stack, or length of the stack, is variable. Items may only be added to or deleted from the top of the stack. For this reason, a stack is also known as a *pushdown list* or a *last-in-first-out (LIFO) list*.

The implementation of a stack requires that there be some set of locations used to store the stack elements. A typical approach is illustrated in Figure P.1. A contiguous block of locations is reserved in main memory (or virtual memory) for the stack. Most of the time, the block is partially filled with stack elements and the remainder is available for stack growth. Three addresses are needed for proper operation, and these are often stored in processor registers:

- **Stack pointer:** Contains the address of the current top of the stack. If an item is appended to (PUSH) or deleted from (POP) the stack, the pointer is decremented or incremented to contain the address of the new top of the stack.

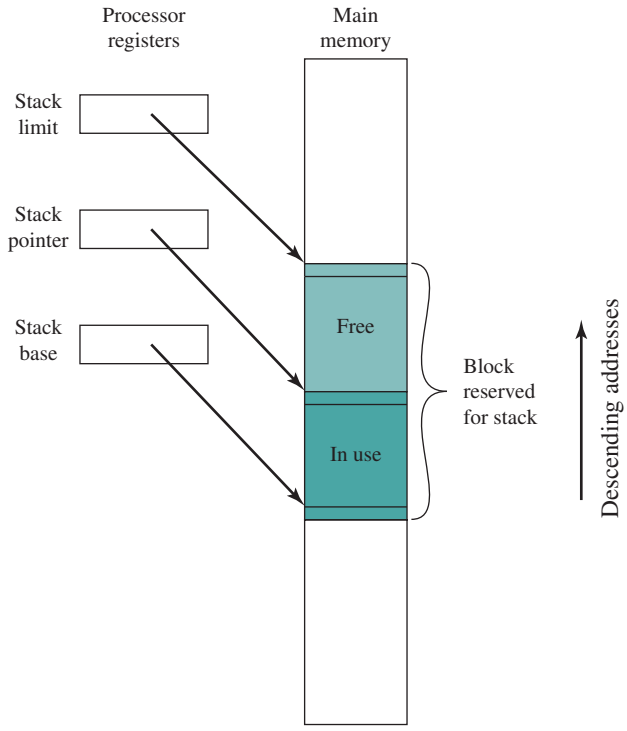


Figure P.1 Typical Stack Organization (full/descending)

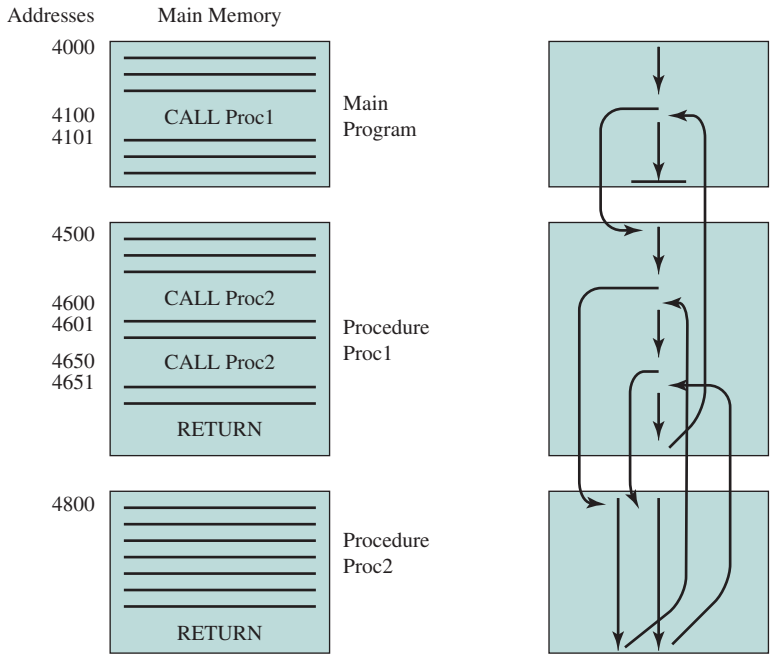
- **Stack base:** Contains the address of the bottom location in the reserved block. This is the first location to be used when an item is added to an empty stack. If an attempt is made to POP an element when the stack is empty, an error is reported.
- **Stack limit:** Contains the address of the other end, or top, of the reserved block. If an attempt is made to PUSH an element when the stack is full, an error is reported.

Traditionally, and on most processors today, the base of the stack is at the high-address end of the reserved stack block, and the limit is at the low-address end. Thus, the stack grows from higher addresses to lower addresses.

P.2 PROCEDURE CALLS AND RETURNS

A common technique for managing procedure calls and returns makes use of a stack. When the processor executes a call, it places (pushes) the return address on the stack. When it executes a return, it uses the address on top of the stack and removes (pops) that address from the stack. For the nested procedures of Figure P.2, Figure P.3 illustrates the use of a stack.

It is also often necessary to pass parameters with a procedure call. These could be passed in registers. Another possibility is to store the parameters in memory just



(a) Calls and returns (b) Execution sequence

Figure P.2 Nested Procedures

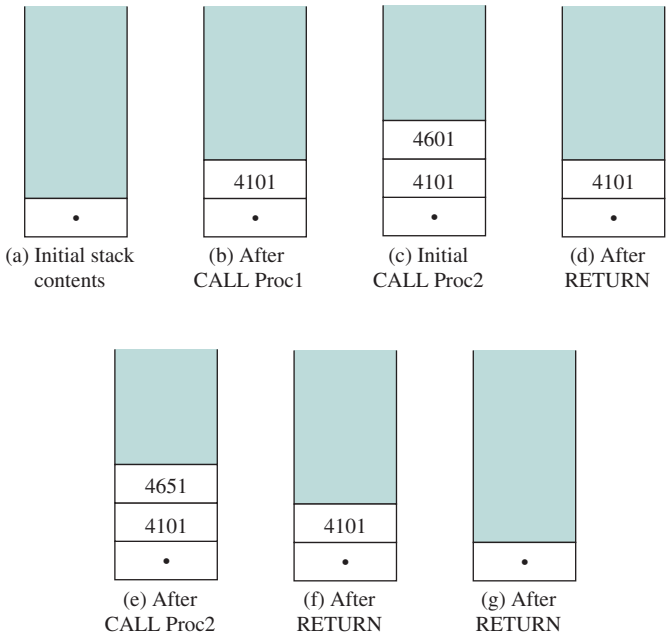


Figure P.3 Use of Stack to Implement Nested Procedures of Figure P.2

after the Call instruction. In this case, the return must be to the location following the parameters. Both of these approaches have drawbacks. If registers are used, the called program and the calling program must be written to assure that the registers are used properly. The storing of parameters in memory makes it difficult to exchange a variable number of parameters.

A more flexible approach to parameter passing is the stack. When the processor executes a call, it not only stacks the return address, it stacks parameters to be passed to the called procedure. The called procedure can access the parameters from the stack. Upon return, return parameters can also be placed on the stack, *under* the return address. The entire set of parameters, including return address, that is stored for a procedure invocation is referred to as a **stack frame**.

An example is provided in Figure P.4. The example refers to procedure P in which the local variables $x1$ and $x2$ are declared, and procedure Q, which can be called by P and in which the local variables $y1$ and $y2$ are declared. The first item stored in each stack frame is a pointer to the beginning of the previous frame. This is needed if the number or length of parameters to be stacked is variable. Next is stored the return point for the procedure that corresponds to this stack frame. Finally, space is allocated at the top of the stack frame for local variables. These local variables can be used for parameter passing. For example, suppose when P calls Q, it passes one parameter value. This value could be stored in variable $y1$. Thus, in a high-level language, there would be an instruction in the P routine that looks like this:

CALL Q($y1$)

When this call is executed, a new stack frame is created for Q (see Figure P.4b), which includes a pointer to the stack frame for P, the return address to P, and two local

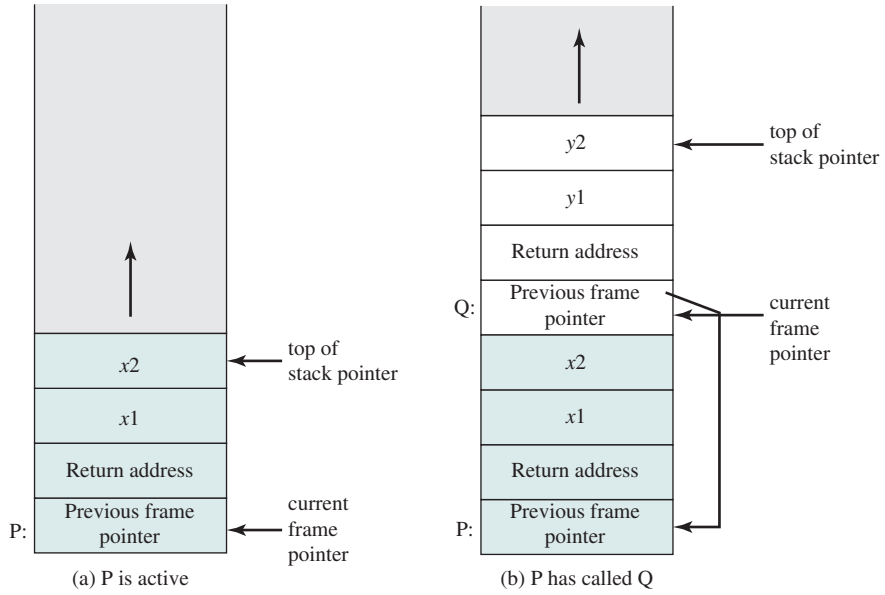


Figure P.4 Stack Frame Growth Using Sample Procedures P and Q

variables for Q, one of which is initialized to the passed parameter value from P. The other local variable, *y*2, is simply a local variable used by Q in its calculations. The need to include such local variables in the stack frame is discussed in the next subsection.

P.3 REENTRANT PROCEDURES

A useful concept, particularly in a system that supports multiple users at the same time, is that of the reentrant procedure. A reentrant procedure is one in which a single copy of the program code can be shared by multiple users during the same period of time. Reentrancy has two key aspects: The program code cannot modify itself and the local data for each user must be stored separately. A reentrant procedure can be interrupted and called by an interrupting program and still execute correctly upon return to the procedure. In a shared system, reentrancy allows more efficient use of main memory: One copy of the program code is kept in main memory, but more than one application can call the procedure.

Thus, a reentrant procedure must have a permanent part (the instructions that make up the procedure) and a temporary part (a pointer back to the calling program as well as memory for local variables used by the program). Each execution instance, called activation, of a procedure will execute the code in the permanent part but must have its own copy of local variables and parameters. The temporary part associated with a particular activation is referred to as an *activation record*.

The most convenient way to support reentrant procedures is by means of a stack. When a reentrant procedure is called, the activation record of the procedure can be stored on the stack. Thus, the activation record becomes part of the stack frame that is created on procedure call.

APPENDIX Q

ECOS

The Embedded Configurable Operating System (eCos) is an open source, royalty-free, real-time OS intended for embedded applications. The system is targeted at high-performance small embedded systems. For such systems, an embedded form of Linux or other commercial OS would not provide the streamlined software required. The eCos software has been implemented on a wide variety of processor platforms, including Intel IA32, PowerPC, SPARC, ARM, CalmRISC, MIPS, and NEC V8xx. It is one of the most widely used embedded operating systems. It is implemented in C/C++.

Q.1 CONFIGURABILITY

An embedded OS that is flexible enough to be used in a wide variety of embedded applications and on a wide variety of embedded platforms must provide more functionality than will be needed for any particular application and platform. For example, many real-time operating systems support task switching, concurrency controls, and a variety of priority scheduling mechanisms. A relatively simple embedded system would not need all these features.

The challenge is to provide an efficient, user-friendly mechanism for configuring selected components and for enabling and disabling particular features within components. The eCos configuration tool, which runs on Windows or Linux, is used to configure an eCos package to run on a target embedded system. The complete eCos package is structured hierarchically, making it easy (using the configuration tool) to assemble a target configuration. At a top level, eCos consists of a number of components, and the configuration user may select only those components needed for the target application. For example, a system might have a particular serial I/O device. The configuration user would select serial I/O for this configuration, then select one or more specific I/O devices to be supported. The configuration tool would include the minimum necessary software for that support. The configuration user can also select specific parameters, such as default data rate and the size of I/O buffers to be used.

This configuration process can be extended down to finer levels of detail, even to the level of individual lines of code. For example, the configuration tool provides the option of including or omitting a priority inheritance protocol.

Figure Q.1 shows the top level of the eCos configuration tool as seen by the tool user. Each of the items on the list in the left-hand window can be selected or deselected. When an item is highlighted, the lower right-hand window provides a description, and the upper right-hand window provides a link to further documentation plus additional information about the highlighted item. Items on the list can be expanded to provide a finer-grained menu of options. Figure Q.2 illustrates an expansion of the eCos kernel option. In this figure, note exception handling has been selected for inclusion, but SMP (symmetric multiprocessing) has been omitted. In general, components and individual options can be selected or omitted. In some cases, individual values can be set; for example, a minimum acceptable stack size is an integer value that can be set or left to a default value.

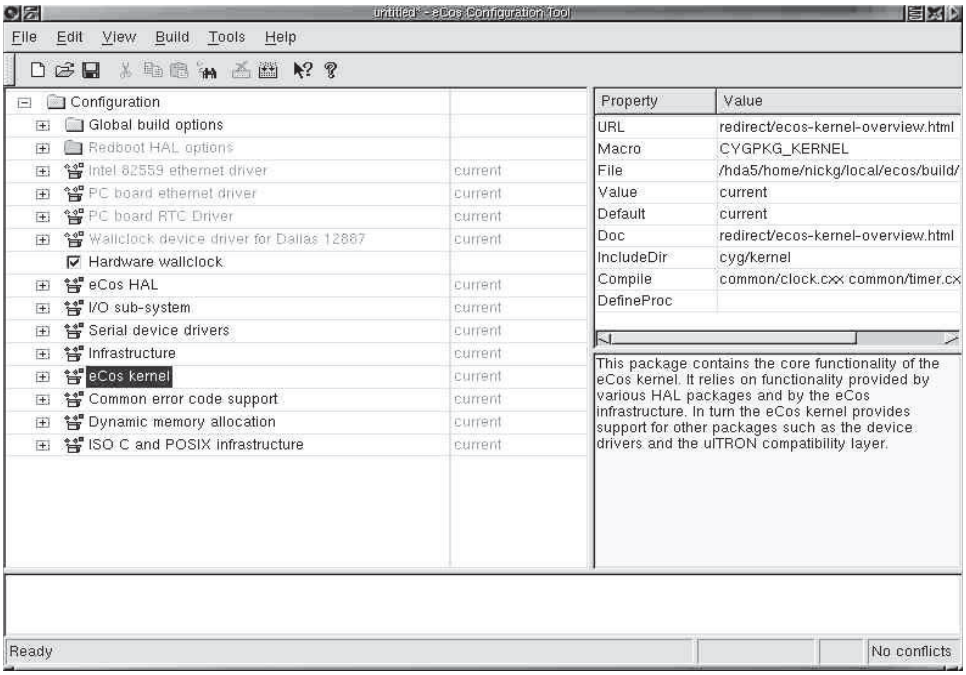


Figure Q.1 **eCos Configuration Tool - Top Level. Courtesy of eCosCentric Limited. Used with permission.**

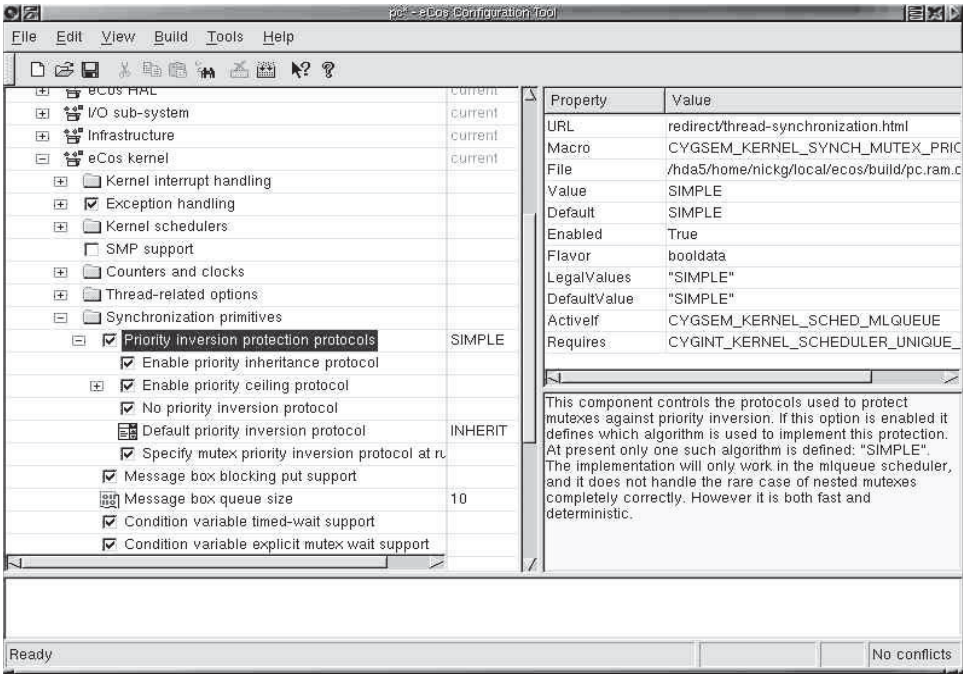


Figure Q.2 **eCos Configuration Tool - Kernel Details. Courtesy of eCosCentric Limited. Used with permission.**

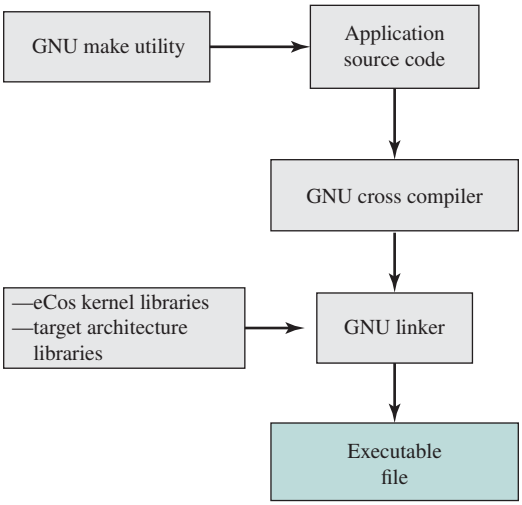


Figure Q.3 Loading an eCos Configuration

Figure Q.3 shows a typical example of the overall process of creating the binary image to execute in the embedded system. This process is run on a source system, such as a Windows or Linux platform, and the executable image is destined to execute on a target embedded system, such as a sensor in an industrial environment. At the highest software level is the application source code for the particular embedded application. This code is independent of eCos but makes use of application programming interfaces (API) to sit on top of the eCos software. There may be only one version of the application source code, or there may be variations for different versions of the target embedded platform. In this example, the GNU make utility is used to selectively determine which pieces of a program need to be compiled or recompiled (in the case of a modified version of the source code) and issues the commands to recompile them. The GNU cross compiler, executing on the source platform, then generates the binary executable code for the target embedded platform. The GNU linker links the application object code with the code generated by the eCos configuration tool. This latter set of software includes selected portions of the eCos kernel plus selected software for the target embedded system. The result can then be loaded into the target system.

Q.2 ECOS COMPONENTS

A key design requirement for eCos is portability to different architectures and platforms with minimal effort. To meet this requirement, eCos consists of a layered set of components (see Figure Q.4).

Hardware Abstraction Layer (HAL)

At the bottom is the hardware abstraction layer (HAL). The HAL is software that presents a consistent API to the upper layers and maps upper-layer operations onto a specific hardware platform. Thus, the HAL is different for each hardware

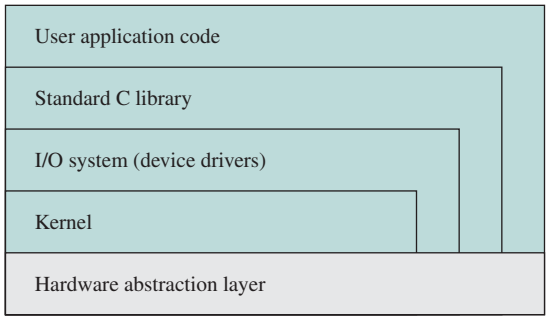


Figure Q.4 eCos Layered Structure

platform. Figure Q.5 is an example that demonstrates how the HAL abstracts hardware-specific implementations for the same API call on two different platforms. As this example shows, the call from an upper layer to enable interrupts is the same on both platforms, but the C code implementation of the function is specific to each platform.

```
1 #define HAL_ENABLE_INTERRUPTS()          \
2     asm volatile (                       \
3         "mrs r3, cpsr;"                  \
4         "bic r3, r3, #0xC0;"             \
5         "mrs cpsr, r3;"                  \
6         :                                \
7         :                                \
8         :  "r3"                          \
9         );                               \
```

(a) ARM architecture

```
1 #define HAL_ENABLE_INTERRUPTS()          \
2     CYG_MACRO_START                      \
3     cyg_uint32 tmp1, tmp2                \
4     asm volatile (                       \
5         "mfmsr   %0;"                    \
6         "ori     %1,%1,0x800;"           \
7         "rlwimi  %0,%1,0,16,16;"         \
8         "mtmsr   %0;"                    \
9         : "=r" (tmp1), "=r" (tmp2));     \
10    CYG_MACRO_END                        \
```

(b) PowerPC architecture

Figure Q.5 Two Implementations of HAL_ENABLE_INTERRUPTS() Macro

The HAL is implemented as three separate modules:

- **Architecture:** Defines the processor family type. This module contains the code necessary for processor startup, interrupt delivery, context switching, and other functionality specific to the instruction set architecture of that processor family.
- **Variant:** Supports the features of the specific processor in the family. An example of a supported feature is an on-chip module such as a memory management unit (MMU).
- **Platform:** Extends the HAL support to tightly coupled peripherals such as interrupt controllers and timer devices. This module defines the platform or board that includes the selected processor architecture and variant. It includes code for startup, chip selection configuration, interrupt controllers, and timer devices.

Note the HAL interface can be directly used by any of the upper layers, promoting efficient code.

eCos Kernel

The eCos kernel was designed to satisfy four main objectives:

- **Low interrupt latency:** The time it takes to respond to an interrupt and begin executing an ISR.
- **Low task switching latency:** The time it takes from when a thread becomes available to when actual execution begins.
- **Small memory footprint:** Memory resources for both program and data are kept to a minimum by allowing all components to configure memory as needed.
- **Deterministic behavior:** Throughout all aspect of execution, the kernels performance must be predictable and bounded to meet real-time application requirements.

The eCos kernel provides the core functionality needed for developing multithreaded applications:

1. The ability to create new threads in the system, either during startup or when the system is already running
2. Control over the various threads in the system: for example, manipulating their priorities
3. A choice of schedulers, determining which thread should currently be running
4. A range of synchronization primitives, allowing threads to interact and share data safely
5. Integration with the system's support for interrupts and exceptions

Some functionality that is typically included in the kernel of an OS is not included in the eCos kernel. For example, memory allocation is handled by a separate package. Similarly, each device driver is a separate package. Various packages are

combined and configured using the eCos configuration technology to meet the requirements of the application. This makes for a lean kernel. Further, the minimal nature of the kernel means that for some embedded platforms, the eCos kernel is not used at all. Simple single-threaded applications can be run directly on HAL. Such configurations can incorporate needed C library functions and device drivers, but avoid the space and time overhead of the kernel.

There are two different techniques for utilizing kernel functions in eCos. One way to employ kernel functionality is by using the C API of kernel. Examples of such functions are `cyg_thread_create` and `cyg_mutex_lock`. These functions can be invoked directly from application code. On the other hand, kernel functions can also be invoked by using compatibility packages for existing API's, for example, POSIX threads or μITRON. The compatibility packages allow application code to call standard functions like `pthread_create`, and those functions are implemented using the basic functions provided by the eCos kernel. Code sharing and reusability of already developed code is easily achieved by use of compatibility packages.

I/O System

The eCos I/O system is a framework for supporting device drivers. A variety of drivers for a variety of platforms are provided in the eCos configuration package. These include drivers for serial devices, Ethernet, flash memory interfaces, and various I/O interconnects such as PCI (Peripheral Component Interconnect) and USB (Universal Serial Bus). In addition, users can develop their own device drivers.

The principal objective for the I/O system is efficiency, with no unnecessary software layering or extraneous functionality. Device drivers provide the necessary functions for input, output, buffering, and device control.

As mentioned, device drivers and other higher-layer software may be implemented directly on the HAL if this is appropriate. If specialized kernel-type functions are needed, then the device driver is implemented using kernel APIs. The kernel provides a three-level interrupt model:

- **Interrupt service routines (ISRs):** Invoked in response to a hardware interrupt. Hardware interrupts are delivered with minimal intervention to an ISR. The HAL decodes the hardware source of the interrupt and calls the ISR of the attached interrupt object. This ISR may manipulate the hardware but is only allowed to make a restricted set of calls on the driver API. When it returns, an ISR may request that its deferred service routine (DSR) should be scheduled to run.
- **Deferred service routines (DSRs):** Invoked in response to a request by an ISR. A DSR will be run when it is safe to do so without interfering with the scheduler. Most of the time the DSR will run immediately after the ISR, but if the current thread is in the scheduler, it will be delayed until the thread is finished. A DSR is allowed to make a larger set of driver API calls, including, in particular, being able to call `cyg_drv_cond_signal()` to wake up waiting threads.
- **Threads:** The clients of the driver. Threads are able to make all API calls and in particular are allowed to wait on mutexes and condition variables.

Tables Q.1 and Q.2 show the device driver interface to the kernel. These tables give a good feel for the type of functionality available in the kernel to support device

Table Q.1 Device Driver Interface to the eCos Kernel: Concurrency

`cyg_drv_spinlock_init` Initialize a spinlock in a locked or unlocked state. `cyg_drv_spinlock_destroy` Destroy a spinlock that is no longer of use. `cyg_drv_spinlock_spin` Claim a spinlock, waiting in a busy loop until it is available. `cyg_drv_spinlock_clear` Clear a spinlock. This clears the spinlock and allows another CPU to claim it. If there is more than one CPU waiting in `cyg_drv_spinlock_spin`, then just one of them will be allowed to proceed. `cyg_drv_spinlock_test` Inspect the state of the spinlock. If the spinlock is not locked, then the result is TRUE. If it is locked then the result will be FALSE. `cyg_drv_spinlock_spin_intsave` This function behaves like `cyg_drv_spinlock_spin` except that it also disables interrupts before attempting to claim the lock. The current interrupt enable state is saved in *istate. Interrupts remain disabled once the spinlock has been claimed and must be restored by calling `cyg_drv_spinlock_clear_intsave`. Device drivers should use this function to claim and release spinlocks rather than the `non-_intsave()` variants, to ensure proper exclusion with code running on both other CPUs and this CPU.
`cyg_drv_mutex_init` Initialize a mutex. `cyg_drv_mutex_destroy` Destroy a mutex. The mutex should be unlocked and there should be no threads waiting to lock it when this call is made. `cyg_drv_mutex_lock` Attempt to lock the mutex pointed to by the mutex argument. If the mutex is already locked by another thread, then this thread will wait until that thread is finished. If the result from this function is FALSE, then the thread was broken out of its wait by some other thread. In this case the mutex will not have been locked. `cyg_drv_mutex_trylock` Attempt to lock the mutex pointed to by the mutex argument without waiting. If the mutex is already locked by some other thread then this function returns FALSE. If the function can lock the mutex without waiting, then TRUE is returned. `cyg_drv_mutex_unlock` Unlock the mutex pointed to by the mutex argument. If there are any threads waiting to claim the lock, one of them is woken up to try and claim it. `cyg_drv_mutex_release` Release all threads waiting on the mutex.
`cyg_drv_cond_init` Initialize a condition variable associated with a mutex. A thread may only wait on this condition variable when it has already locked the associated mutex. Waiting will cause the mutex to be unlocked, and when the thread is reawakened, it will automatically claim the mutex before continuing. `cyg_drv_cond_destroy` Destroy the condition variable. `cyg_drv_cond_wait` Wait for a signal on a condition variable. `cyg_drv_cond_signal` Signal a condition variable. If there are any threads waiting on this variable, at least one of them will all be awakened. `cyg_drv_cond_broadcast` Signal a condition variable. If there are any threads waiting on this variable, they will all be awakened.

Table Q.2 Device Driver Interface to the eCos Kernel: Interrupts

`cyg_drv_isr_lock` Disable delivery of interrupts, preventing all ISRs running. This function maintains a counter of the number of times it is called. `cyg_drv_isr_unlock` Reenable delivery of interrupts, allowing ISRs to run. This function decrements the counter maintained by `cyg_drv_isr_lock`, and only reallows interrupts when it goes to zero. `cyg_ISR_t` Define ISR.
`cyg_drv_dsr_lock` Disable scheduling of DSRs. This function maintains a counter of the number of times it has been called. `cyg_drv_dsr_unlock` Reenable scheduling of DSRs. This function decrements the counter incremented by `cyg_drv_dsr_lock`. DSRs are only allowed to be delivered when the counter goes to zero. `cyg_DSR_t Define` DSR prototype.
`cyg_drv_interrupt_create` Create an interrupt object and returns a handle to it. `cyg_drv_interrupt_delete` Detach the interrupt from the vector and free the memory for reuse. `cyg_drv_interrupt_attach` Attach an interrupt to a vector so that interrupts will be delivered to the ISR when the interrupt occurs. `cyg_drv_interrupt_detach` Detach the interrupt from the vector so that interrupts will no longer be delivered to the ISR. `cyg_drv_interrupt_mask` Program the interrupt controller to stop delivery of interrupts on the given vector. `cyg_drv_interrupt_mask_intunsafe` Program the interrupt controller to stop delivery of interrupts on the given vector. This version differs from `cyg_drv_interrupt_mask` in not being interrupt safe. So in situations where, for example, interrupts are already known to be disabled, this may be called to avoid the extra overhead. `cyg_drv_interrupt_unmask`, `cyg_drv_interrupt_unmask_intunsafe` Program the interrupt controller to reallow delivery of interrupts on the given vector. `cyg_drv_interrupt_acknowledge` Perform any processing required at the interrupt controller and in the CPU to cancel the current interrupt request. `cyg_drv_interrupt_configure` Program the interrupt controller with the characteristics of the interrupt source. `cyg_drv_interrupt_level` Program the interrupt controller to deliver the given interrupt at the supplied priority level. `cyg_drv_interrupt_set_cpu` On multiprocessor systems, this function causes all interrupts on the given vector to be routed to the specified CPU. Subsequently, all such interrupts will be handled by that CPU. `cyg_drv_interrupt_get_cpu` On multiprocessor systems, this function returns the ID of the CPU to which interrupts on the given vector are currently being delivered.

drivers. Note the device driver interface can be configured for one or more of the following concurrency mechanisms: spinlocks, condition variables, and mutexes. These are described in a subsequent portion of this discussion.

Standard C Libraries

A complete Standard C run-time library is provided. Also included is a complete math run-time library for high-level mathematics functions, including a complete IEEE-754 floating-point library for those platforms without hardware floating points.

Q.3 ECOS SCHEDULER

The eCos kernel can be configured to provide one of two scheduler designs: the bitmap scheduler and a multilevel queue scheduler. The configuration user selects the appropriate scheduler for the environment and the application. The bitmap scheduler provides efficient scheduling for a system with a small number of threads that may be active at any point in time. The multiqueue scheduler is appropriate if the number of threads is dynamic or if it is desirable to have multiple threads at the same priority level. The multilevel scheduler is also needed if time slicing is desired.

Bitmap Scheduler

A bitmap scheduler supports multiple priority levels, but only one thread can exist at each priority level at any given time. Scheduling decisions are quite simple with this scheduler (see Figure Q.6a). When a blocked thread become ready to run, it may preempt a thread of lower priority. When a running thread suspends, the ready thread with the highest priority is dispatched. A thread can be suspended because it is blocked on a synchronization primitive, because it is interrupted, or because it relinquishes control. Because there is only one thread, at most, at each priority level, the scheduler does not have to make a decision as to which thread at a given priority level should be dispatched next.

The bitmap scheduler is configured with 8, 16, or 32 priority levels. A simple bitmap is kept of the threads that are ready to execute. The scheduler need only to determine the position of the most significant one bit in the bitmap to make a scheduling decision.

Multilevel Queue Scheduler

As with the bitmap scheduler, the multilevel queue scheduler supports up to 32 priority levels. The multilevel queue scheduler allows for multiple active threads at each priority level, limited only by system resources.

Figure Q.6b illustrates the nature of the multilevel queue scheduler. A data structure represents the number of ready threads at each priority level. When a blocked thread become ready to run, it may preempt a thread of lower priority. As with the bitmap scheduler, a running thread may be blocked on a synchronization primitive, because it is interrupted, or because it relinquishes control. When

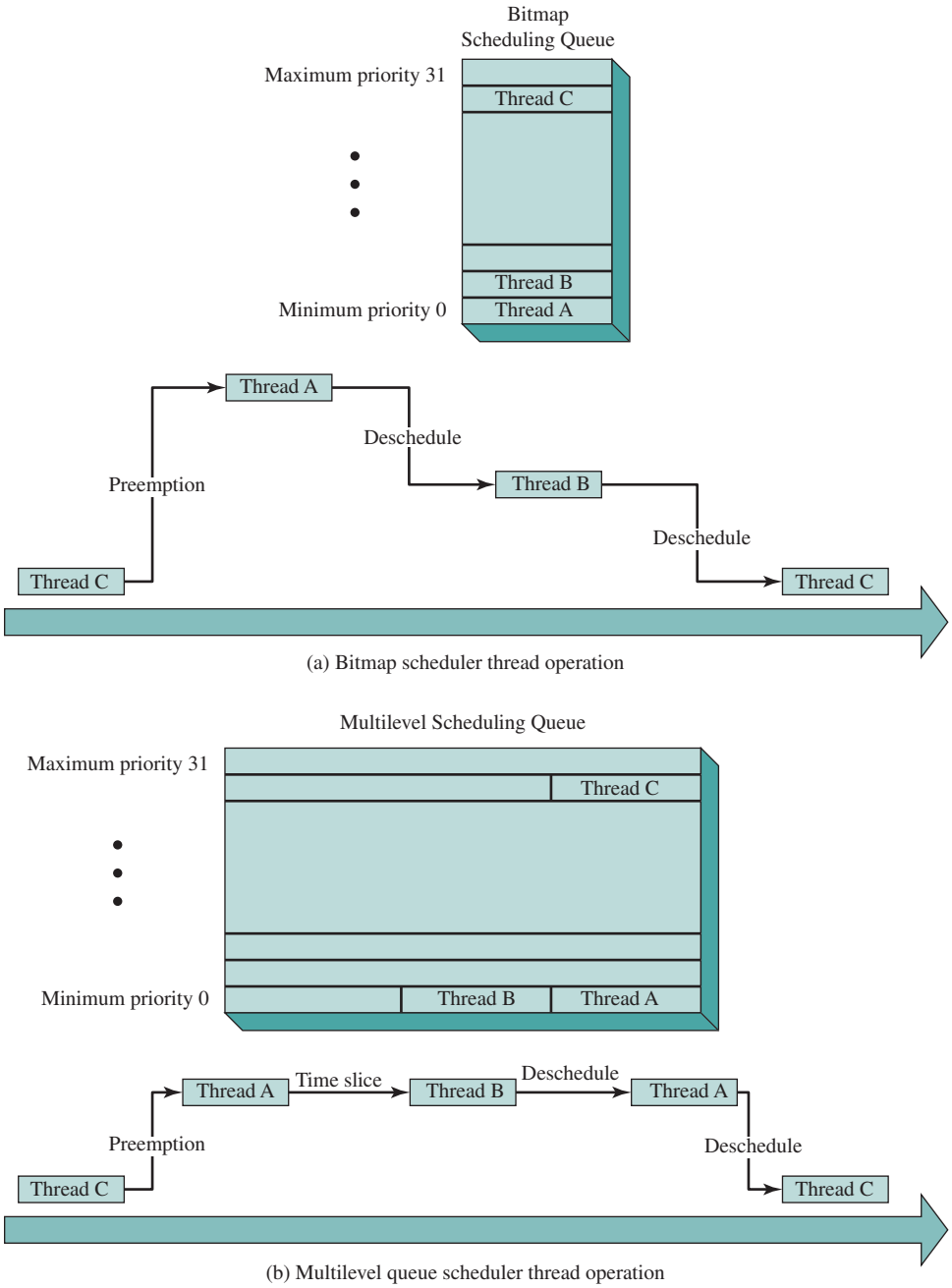


Figure Q.6 eCos Scheduler Options

a thread is blocked, the scheduler must first determine if one or more threads at the same priority level as the blocked thread is ready. If so, the scheduler chooses the one at the front of the queue. Otherwise, the scheduler looks for the next highest priority level with one or more ready threads and dispatches one of these threads.

In addition, the multilevel queue scheduler can be configured for time slicing. Thus, if a thread is running and there is one or more ready threads at the same priority level, the scheduler will suspend the running thread after one time slice and choose the next thread in the queue at that priority level. This is a round-robin policy within one priority level. Not all applications require time slicing. For example, an application may contain only threads that block regularly for some other reason. For these applications, the user can disable time slicing, which reduces the overhead associated with timer interrupts.

Q.4 ECOS THREAD SYNCHRONIZATION

The eCos kernel can be configured to include one or more of six different thread synchronization mechanisms. These include the classic synchronization mechanisms: mutexes, semaphores, and condition variables. In addition, eCos supports two synchronization/communication mechanisms that are common in real-time systems, namely event flags and mailboxes. Finally, the eCos kernel supports spinlocks, which are useful in SMP (symmetric multiprocessing) systems.

Mutexes

The mutex (mutual exclusion lock) was introduced in Chapter 6. Recall that a mutex is used to enforce mutually exclusive access to a resource, allowing only one thread at a time to gain access. The mutex has only two states: locked and unlocked. This is similar to a binary semaphore: When a mutex is locked by one thread, any other thread attempting to lock the mutex is blocked; when the mutex is unlocked, then one of the threads blocked on this mutex is unblocked and allowed to lock the mutex and gain access to the resource.

The mutex differs from a binary semaphore in two respects. First, the thread that locks the mutex must be the one to unlock it. In contrast, it is possible for one thread to lock a binary semaphore and for another to unlock it. The other difference is that a mutex provides protection against priority inversion, whereas a semaphore does not.

The eCos kernel can be configured to support either a priority inheritance protocol or a priority ceiling protocol. These are described in Chapter 10.

Semaphores

The eCos kernel provides support for a counting semaphore. Recall from Chapter 5 that a counting semaphore is an integer value used for signaling among threads. The `cyg_semaphore_init` is used to initialize a semaphore. The `cyg_semaphore_post` command increments the semaphore count when an event occurs. If

the new count is less than or equal to zero, then a thread is waiting on this semaphore and is awakened. The `cyg_semaphore_wait` function checks the value of a semaphore count. If the count is zero, the thread calling this function will wait for the semaphore. If the count is nonzero, the count is decremented and the thread continues.

Counting semaphores are suited to enabling threads to wait until an event has occurred. The event may be generated by a producer thread, or by a DSR in response to a hardware interrupt. Associated with each semaphore is an integer counter that keeps track of the number of events that have not yet been processed. If this counter is zero, an attempt by a consumer thread to wait on the semaphore will block until some other thread or a DSR posts a new event to the semaphore. If the counter is greater than zero, then an attempt to wait on the semaphore will consume one event (in other words decrement the counter) and return immediately. Posting to a semaphore will wake up the first thread that is currently waiting, which will then resume inside the semaphore wait operation and decrement the counter again.

Another use of semaphores is for certain forms of resource management. The counter would correspond to how many of a certain type of resource are currently available, with threads waiting on the semaphore to claim a resource and posting to release the resource again. In practice, condition variables are usually much better suited for operations like this.

Condition Variables

A condition variable is used to block a thread until a particular condition is true. Condition variables are used with mutexes to allow multiple threads to access shared data. They can be used to implement monitors of the type discussed in Chapter 6 (e.g., Figure 6.14). The basic commands are as follows:

`cyg_cond_wait` Causes the current thread to wait on the specified condition variable and simultaneously unlocks the mutex attached to the condition variable.

`cyg_cond_signal` Wakes up one of the threads waiting on this condition variable, causing that thread to become the owner of the mutex.

`cyg_cond_broadcast` Wakes up all the threads waiting on this condition variable. Each thread that was waiting on the condition variable becomes the owner of the mutex when it runs.

In eCos, condition variables are typically used in conjunction with mutexes to implement long-term waits for some condition to become true. Consider the following example. Figure Q.7 defines a set of functions to control access to a pool of resources using mutexes. The mutex is used to make the allocation and freeing of resources from a pool atomic. The function `res_t res_allocate` checks to see if one or more units of a resource are available and, if so, takes one unit. This operation is protected by a mutex so no other thread can check or alter the resource pool while this thread has control of the mutex. The function `res_free(res_t res)` enables a thread to release one unit of a resource that it had previously acquired. Again, this operation is made atomic by a mutex.

In this example, if a thread attempts to access a resource and none are available, the function returns `RES_NONE`. Suppose, however, we want the thread to be blocked and wait for a resource to become available, rather than returning `RES_NONE`.

```
cyg_mutex_t res_lock;
res_t res_pool[RES_MAX];
int res_count = RES_MAX;

void res_init(void)
{
    cyg_mutex_init(&res_lock);
     <fill pool with resources>
}
res_t res_allocate(void)
{
    res_t res;
    cyg_mutex_lock(&res_lock);       // lock the mutex
    if( res_count == 0 )             // check for free resource
        res =RES_NONE;               // return RES_NONE if none
    else  {
        res_count--;                 // allocate a resources
        res =res_pool[res_count];
}
    cyg_mutex_unlock(&res_lock);     // unlock the mutex
    return res;
}
void res_free(res_t res)
{
    cyg_mutex_lock(&res_lock);       // lock the mutex
    res_pool[res_count] =res;        // free the resource
    res_count++;
    cyg_mutex_unlock(&res_lock);     // unlock the mutex
}
```

Figure Q.7 Controlling Access to a Pool of Resources Using Mutexes

Figure Q.8 accomplishes this with the use of a condition variable associated with the mutex. When `res_allocate` detects that there are no resources, it calls `cyg_cond_wait`. This latter function unlocks the mutex and puts the calling thread to sleep on the condition variable. When `res_free` is eventually called, it puts a resource back into the pool and calls `cyg_cond_signal` to wake up any thread waiting on the condition variable. When the waiting thread eventually gets to run again, it will relock the mutex before returning from `cyg_cond_wait`.

There are two significant features of this example, and of the use of condition variables in general. First, the mutex unlock and wait in `cyg_cond_wait` are atomic: No other thread can run between the unlock and the wait. If this were not the case, then a call to `res_free` by some other thread would release the resource, but the call to `cyg_cond_signal` would be lost, and the first thread would end up waiting when there were resources available.

```
cyg_mutex_t res_lock;
cyg_cond_t res_wait;
res_t res_pool[RES_MAX];
int res_count =RES_MAX;
void res_init(void)
{
    cyg_mutex_init(&res_lock);
    cyg_cond_init(&res_wait, &res_lock);
    <fill pool with resources>
}
res_t res_allocate(void)
{
    res_t res;
    cyg_mutex_lock(&res_lock);        // lock the mutex
    while( res_count == 0 )           // wait for a resources
       cyg_cond_wait(&res_wait);
    res_count--;                      // allocate a resource
    res =res_pool[res_count];
    cyg_mutex_unlock(&res_lock);      // unlock the mutex
    return res;
}
void res_free(res_t res)
{
    cyg_mutex_lock(&res_lock);         // lock the mutex
    res_pool[res_count] =res;          // free the resource
    res_count++;
    cyg_cond_signal(&res_wait);        // wake up any waiting
                                          allocators
    cyg_mutex_unlock(&res_lock);       // unlock the mutex
}
```

Figure Q.8 Controlling Access to a Pool of Resources Using Mutexes and Condition Variables

The second feature is that the call to `cyg_cond_wait` is in a `while` loop and not a simple `if` statement. This is because of the need to relock the mutex in `cyg_cond_wait` when the signaled thread reawakens. If there are other threads already queued to claim the lock, then this thread must wait. Depending on the scheduler and the queue order, many other threads may have entered the critical section before this one gets to run. So the condition that it was waiting for may have been rendered false. Using a loop around all condition variable wait operations is the only way to guarantee that the condition being waited for is still true after waiting.

Event Flags

An event flag is a 32-bit word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the

corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR). A signaling thread can set or reset bits based on specific conditions or events so that the appropriate thread is unblocked. For example, bit 0 could represent completion of a specific I/O operation, making data available, and bit 1 could indicate that the user has pressed a start button. A producer thread or DSR could set these two bits, and a consumer thread waiting on these two events will be woken up.

A thread can wait on one or more events using the `cyg_flag_wait` command, which takes three arguments: a particular event flag, a combination of bit positions in the flag, and a mode parameter. The mode parameter specifies whether the thread will block until all the bits are set (AND) or until at least one of the bits is set (OR). The mode parameter may also specify that when the wait succeeds, the entire event flag is cleared (set to all zeros).

Mailboxes

Mailboxes, also called message boxes, are an eCos synchronization mechanism that provides a means for two threads to exchange information. Section 5.5 provides a general discussion of message-passing synchronization. Here, we look at the specifics of the eCos version.

The eCos mailbox mechanism can be configured for blocking or nonblocking on both the send and receive side. The maximum size of the message queue associated with a given mailbox can also be configured.

The send message primitive, called put, includes two arguments: a handle to the mailbox and a pointer for the message itself. There are three variants to this primitive:

`cyg_mbox_put` If there is a spare slot in the mailbox, then the new message is placed there; if there is a waiting thread, it will be woken up so it can receive the message. If the mailbox is currently full, `cyg_mbox_put` blocks until there has been a corresponding get operation and a slot is available.

`cyg_mbox_timed_put` Same as `cyg_mbox_put` if there is a spare slot. Otherwise, the function will wait a specified time limit and place the message if a slot becomes available. If the time limit expires, the operation returns false. Thus, `cyg_mbox_timed_put` is blocking only for less than or equal to a specified time interval.

`cyg_mbox_tryput` This is a nonblocking version, which returns true if the message is sent successfully and false if the mailbox is full.

Similarly, there are three variants to the get primitive.

`cyg_mbox_get` If there is a pending message in the specified mailbox, `cyg_mbox_get` returns with the message that was put into the mailbox. Otherwise this function blocks until there is a put operation.

`cyg_mbox_timed_get` Immediately returns a message if one is available. Otherwise, the function will wait until either a message is available or until a number of clock ticks have occurred. If the time limit expires, the operation returns a null pointer. Thus, `cyg_mbox_timed_get` is blocking only for less than or equal to a specified time interval.

`cyg_mbox_tryget` This is a nonblocking version, which returns a message if one is available and a null pointer if the mailbox is empty.

Spinlocks

A spinlock is a flag that a thread can check before executing a particular piece of code. Recall from our discussion of Linux spinlocks in Chapter 6 the basic operation of the spinlock: Only one thread at a time can acquire a spinlock. Any other thread attempting to acquire the same lock will keep trying (spinning) until it can acquire the lock. In essence, a spinlock is built on an integer location in memory that is checked by each thread before it enters its critical section. If the value is 0, the thread sets the value to 1 and enters its critical section. If the value is nonzero, the thread continually checks the value until it is zero.

A spinlock should not be used on a single-processor system, which is why it is compiled away on Linux. As an example of the danger, consider a uniprocessor system with preemptive scheduling, in which a higher-priority thread attempts to acquire a spinlock already held by a lower-priority thread. The lower-priority thread cannot execute so as to finish its work and release the spinlock, because the higher-priority thread preempts it. The higher-priority thread can execute but is stuck checking the spinlock. As a result, the higher-priority thread will just loop forever and the lower-priority thread will never get another chance to run and release the spinlock. On an SMP system, the current owner of a spinlock can continue running on a different processor.

GLOSSARY

access method The method that is used to find a file, a record, or a set of records.

address space The range of addresses available to a computer program.

address translator A functional unit that transforms virtual addresses to real addresses.

application programming interface (API) A standardized library of programming tools used by software developers to write applications that are compatible with a specific operating system or graphic user interface.

asynchronous operation An operation that occurs without a regular or predictable time relationship to a specified event, for example, the calling of an error diagnostic routine that may receive control at any time during the execution of a computer program.

base address An address that is used as the origin in the calculation of addresses in the execution of a computer program.

batch processing Pertaining to the technique of executing a set of computer programs such that each is completed before the next program of the set is started.

Beowulf Defines a class of clustered computing that focuses on minimizing the price-to-performance ratio of the overall system without compromising its ability to perform the computation work for which it is being built. Most Beowulf systems are implemented on Linux computers.

binary semaphore A semaphore that takes on only the values 0 and 1. A binary semaphore allows only one process or thread to have access to a shared critical resource at a time.

block (1) A collection of contiguous records that are recorded as a unit; the units are separated by interblock gaps. (2) A group of bits that are transmitted as a unit.

B-tree A technique for organizing indexes. In order to keep access time to a minimum, it stores the data keys in a balanced hierarchy that continually realigns itself as items are inserted and deleted. Thus, all nodes always have a similar number of keys.

busy waiting The repeated execution of a loop of code while waiting for an event to occur.

cache memory A memory that is smaller and faster than main memory and that is interposed between the processor and main memory. The cache acts as a buffer for recently used memory locations.

central processing unit (CPU) That portion of a computer that fetches and executes instructions. It consists of an Arithmetic and Logic Unit (ALU), a control unit, and registers. Often simply referred to as a *processor*.

chained list A list in which data items may be dispersed but in which each item contains an identifier for locating the next item.

client A process that requests services by sending messages to server processes.

cluster A group of interconnected, whole computers working together as a unified computing resource that can create the illusion of being one machine. The term *whole computer* means a system that can run on its own, apart from the cluster.

communications architecture The hardware and software structure that implements the communications function.

compaction A technique used when memory is divided into variable-size partitions. From time to time, the operating system shifts the partitions so they are contiguous and so all of the free memory is together in one block. See *external fragmentation*.

concurrent Pertaining to processes or threads that take place within a common interval of time during which they may have to alternately share common resources.

consumable resource A resource that can be created (produced) and destroyed (consumed). When a resource is acquired by a process, the resource ceases to exist. Examples of consumable resources are interrupts, signals, messages, and information in I/O buffers.

critical section In an asynchronous procedure of a computer program, a part that cannot be executed simultaneously with an associated critical section of another asynchronous procedure. See *mutual exclusion*.

database A collection of interrelated data, often with controlled redundancy, organized according to a schema to serve one or more applications; the data are stored so they can be used by different programs without concern for the data structure or organization. A common approach is used to add new data, and to modify and retrieve existing data.

deadlock (1) An impasse that occurs when multiple processes are waiting for the availability of a resource that will not become available because it is being held by another process that is in a similar wait state. (2) An impasse that occurs when multiple processes are waiting for an action by or a response from another process that is in a similar wait state.

deadlock avoidance A dynamic technique that examines each new resource request for deadlock. If the new request could lead to a deadlock, then the request is denied.

deadlock detection A technique in which requested resources are always granted when available. Periodically, the operating system tests for deadlock.

deadlock prevention A technique that guarantees that a deadlock will not occur. Prevention is achieved by assuring that one of the necessary conditions for deadlock is not met.

demand paging The transfer of a page from secondary memory to main memory storage at the moment of need. Compare *prepaging*.

device driver An operating system module (usually in the kernel) that deals directly with a device or I/O module.

direct access The capability to obtain data from a storage device or to enter data into a storage device in a sequence independent of their relative position, by means of addresses that indicate the physical location of the data.

direct memory access (DMA) A form of I/O in which a special module, called a DMA module, controls the exchange of data between main memory and an I/O device. The processor sends a request for the transfer of a block of data to the DMA module, and is interrupted only after the entire block has been transferred.

disabled interrupt A condition, usually created by the operating system, during which the processor will ignore interrupt request signals of a specified class.

disk allocation table A table that indicates which blocks on secondary storage are free and available for allocation to files.

disk cache A buffer, usually kept in main memory, that functions as a cache of disk blocks between disk memory and the rest of main memory.

dispatch To allocate time on a processor to jobs or tasks that are ready for execution.

distributed operating system A common operating system shared by a network of computers. The distributed operating system provides support for interprocess communication, process migration, mutual exclusion, and the prevention or detection of deadlock.

dynamic relocation A process that assigns new absolute addresses to a computer program during execution so the program may be executed from a different area of main storage.

enabled interrupt A condition, usually created by the operating system, during which the processor will respond to interrupt request signals of a specified class.

encryption The conversion of plain text or data into unintelligible form by means of a reversible mathematical computation.

execution context Same as *process state*.

external fragmentation Occurs when memory is divided into variable-size partitions corresponding to the blocks of data assigned to the memory (e.g., segments in main memory). As segments are moved into and out of the memory, gaps will occur between the occupied portions of memory.

field (1) Defined logical data that are part of a record. (2) The elementary unit of a record that may contain a data item, a data aggregate, a pointer, or a link.

file A set of related records treated as a unit.

file allocation table (FAT) A table that indicates the physical location on secondary storage of the space allocated to a file. There is one file allocation table for each file.

file management system A set of system software that provides services to users and applications in the use of files, including file access, directory maintenance, and access control.

file organization The physical order of records in a file, as determined by the access method used to store and retrieve them.

first-come-first-served (FCFS) Same as *FIFO*.

first-in-first-out (FIFO) A queueing technique in which the next item to be retrieved is the item that has been in the queue for the longest time.

frame In paged virtual storage, a fixed-length block of main memory that is used to hold one page of virtual memory.

gang scheduling The scheduling of a set of related threads to run on a set of processors at the same time, on a one-to-one basis.

hash file A file in which records are accessed according to the values of a key field. Hashing is used to locate a record on the basis of its key value.

hashing The selection of a storage location for an item of data by calculating the address as a function of the contents of the data. This technique complicates the storage allocation function but results in rapid random retrieval.

hit ratio In a two-level memory, the fraction of all memory accesses that are found in the faster memory (e.g., the cache).

indexed access Pertaining to the organization and accessing of the records of a storage structure through a separate index to the locations of the stored records.

indexed file A file in which records are accessed according to the value of key fields. An index is required that indicates the location of each record on the basis of each key value.

indexed sequential access Pertaining to the organization and accessing of the records of a storage structure through an index of the keys that are stored in arbitrarily partitioned sequential files.

indexed sequential file A file in which records are ordered according to the values of a key field. The main file is supplemented with an index file that contains a partial list of key values; the index provides a lookup capability to quickly reach the vicinity of a desired record.

instruction cycle The time period during which one instruction is fetched from memory and executed when a computer is given an instruction in machine language.

internal fragmentation Occurs when memory is divided into fixed-size partitions (e.g., page frames in main memory, physical blocks on disk). If a block of data is assigned to one or more partitions, then there may be wasted space in the last partition. This will occur if the last portion of data is smaller than the last partition.

interrupt A suspension of a process, such as the execution of a computer program, caused by an event external to that process and performed in such a way that the process can be resumed.

interrupt handler A routine, generally part of the operating system. When an interrupt occurs, control is transferred to the corresponding interrupt handler, which takes some action in response to the condition that caused the interrupt.

job A set of computational steps packaged to run as a unit.

job control language (JCL) A problem-oriented language that is designed to express statements in a job that are used to identify the job or to describe its requirements to an operating system.

kernel A portion of the operating system that includes the most heavily used portions of software. Generally, the kernel is maintained permanently in main memory. The kernel runs in a privileged mode and responds to calls from processes and interrupts from devices.

kernel mode A privileged mode of execution reserved for the kernel of the operating system. Typically, kernel mode allows access to regions of main memory that are unavailable to processes executing in a less-privileged mode, and also enables execution of certain machine instructions that are restricted to the kernel mode. Also referred to as *system mode* or *privileged mode*.

last-in-first-out (LIFO) A queueing technique in which the next item to be retrieved is the item most recently placed in the queue.

lightweight process A thread.

livelock A condition in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work. This is similar to deadlock in that no progress is made, but it differs in that neither process is blocked or waiting for anything.

locality of reference The tendency of a processor to access the same set of memory locations repetitively over a short period of time.

logical address A reference to a memory location independent of the current assignment of data to memory. A translation must be made to a physical address before the memory access can be achieved.

logical record A record independent of its physical environment; portions of one logical record may be located in different physical records or several logical records or parts of logical records may be located in one physical record.

macrokernel A large operating system core that provides a wide range of services.

mailbox A data structure shared among a number of processes that is used as a queue for messages. Messages are sent to the mailbox and retrieved from the mailbox rather than passing directly from sender to receiver.

main memory Memory that is internal to the computer system, is program addressable, and can be loaded into registers for subsequent execution or processing.

malicious software Any software designed to cause damage to or use up the resources of a target computer. Malicious software (malware) is frequently concealed within or masquerades as legitimate software. In some cases, it spreads itself to other computers via e-mail or infected disks. Types of malicious software include viruses, Trojan horses, worms, and hidden software for launching denial-of-service attacks.

memory cycle time The time it takes to read one word from or write one word to memory. This is the inverse of the rate at which words can be read from or written to memory.

memory partitioning The subdividing of storage into independent sections.

message A block of information that may be exchanged between processes as a means of communication.

microkernel A small, privileged operating system core that provides process scheduling, memory management, and communication services and relies on other processes to perform some of the functions traditionally associated with the operating system kernel.

mode switch A hardware operation that occurs that causes the processor to execute in a different mode (kernel or process). When the mode switches from

process to kernel, the program counter, processor status word, and other registers are saved. When the mode switches from kernel to process, this information is restored.

monitor A programming language construct that encapsulates variables, access procedures, and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.

monolithic kernel A large kernel containing virtually the complete operating system, including scheduling, file system, device drivers, and memory management. All the functional components of the kernel have access to all of its internal data structures and routines. Typically, a monolithic kernel is implemented as a single process, with all elements sharing the same address space.

multilevel security A capability that enforces access control across multiple levels of classification of data.

multiprocessing A mode of operation that provides for parallel processing by two or more processors of a multiprocessor.

multiprocessor A computer with two or more processors that have common access to a main storage.

multiprogramming A mode of operation that provides for the interleaved execution of two or more computer programs by a single processor. The same as multitasking, using different terminology.

multiprogramming level The number of processes that are partially or fully resident in main memory.

multithreading Multitasking within a single program. It allows multiple streams of instructions (threads) to execute concurrently within the same program, each stream processing a different transaction or message. Each stream is a "subprocess," and the operating system typically cooperates with the application to handle the threads.

multitasking A mode of operation that provides for the concurrent performance or interleaved execution of two or more computer tasks. The same as multiprogramming, using different terminology.

mutex A programming flag used to grab and release an object. When data are acquired that cannot be shared or processing is started that cannot be performed simultaneously elsewhere in the system, the mutex is set to "lock," which blocks other attempts to use it. The mutex is set to "unlock" when the data are no longer needed or the routine is finished. Similar to a *binary semaphore*. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1). In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.

mutual exclusion A condition in which there is a set of processes, only one of which is able to access a given resource or perform a given function at any time. See *critical section*.

nonprivileged state An execution context that does not allow sensitive hardware instructions to be executed, such as the halt instruction and I/O instructions.

nonuniform memory access (NUMA) multiprocessor A shared-memory multiprocessor in which the access time from a given processor to a word in memory varies with the location of the memory word.

object request broker An entity in an object-oriented system that acts as an intermediary for requests sent from a client to a server.

operating system Software that controls the execution of programs and provides services such as resource allocation, scheduling, input/output control, and data management.

page In virtual storage, a fixed-length block that has a virtual address and is transferred as a unit between main memory and secondary memory.

page fault Occurs when the page containing a referenced word is not in main memory. This causes an interrupt and requires the proper page be brought into main memory.

page frame A fixed-size contiguous block of main memory used to hold a page.

paging The transfer of pages between main memory and secondary memory.

physical address The absolute location of a unit of data in memory (e.g., word or byte in main memory, block on secondary memory).

pipe A circular buffer allowing two processes to communicate on the producer–consumer model. Thus, it is a first-in-first-out queue, written by one process and read by another. In some systems, the pipe is generalized to allow any item in the queue to be selected for consumption.

preemption Reclaiming a resource from a process before the process has finished using it.

prepaging The retrieval of pages other than the one demanded by a page fault. The hope is that the additional pages will be needed in the near future, conserving disk I/O. Compare *demand paging*.

priority inversion A circumstance in which the operating system forces a higher-priority task to wait for a lower-priority task.

privileged instruction An instruction that can be executed only in a specific mode, usually by a supervisory program.

privileged mode Same as *kernel mode*.

process A program in execution. A process is controlled and scheduled by the operating system. Same as *task*.

process control block The manifestation of a process in an operating system. It is a data structure containing information about the characteristics and state of the process.

process descriptor Same as process control block.

process image All of the ingredients of a process, including program, data, stack, and process control block.

process migration The transfer of a sufficient amount of the state of a process from one machine to another for the process to execute on the target machine.

process spawning The creation of a new process by another process.

process state All of the information the operating system needs to manage a process and the processor needs to properly execute the process. The process state includes the contents of the various processor registers, such as the program counter and data registers; it also includes information of use to the operating system, such as the priority of the process and whether the process is waiting for the completion of a particular I/O event. Same as *execution context*.

process switch An operation that switches the processor from one process to another by saving all the process control block, registers, and other information for the first and replacing them with the process information for the second.

processor In a computer, a functional unit that interprets and executes instructions. A processor consists of at least an instruction control unit and an arithmetic unit.

program counter Instruction address register.

program status word (PSW) A register or set of registers that contains condition codes, execution mode, and other status information that reflects the state of a process.

programmed I/O A form of I/O in which the CPU issues an I/O command to an I/O module and must then wait for the operation to be complete before proceeding.

race condition Situation in which multiple processes access and manipulate shared data with the outcome dependent on the relative timing of the processes.

real address A physical address in main memory.

real-time system An operating system that must schedule and manage real-time tasks.

real-time task A task that is executed in connection with some process or function or set of events external to the computer system, and must meet one or more deadlines to interact effectively and correctly with the external environment.

record A group of data elements treated as a unit.

reentrant procedure A routine that may be entered before the completion of a prior execution of the same routine and execute correctly.

registers High-speed memory internal to the CPU. Some registers are user visible; that is, available to the programmer via the machine instruction set. Other registers are used only by the CPU, for control purposes.

relative address An address calculated as a displacement from a base address.

remote procedure call (RPC) A technique by which two programs on different machines interact using procedure call/return syntax and semantics. Both the called and calling program behave as if the partner program were running on the same machine.

rendezvous In message passing, a condition in which both the sender and receiver of a message are blocked until the message is delivered.

resident set That portion of a process that is actually in main memory at a given time. Compare *working set*.

response time In a data system, the elapsed time between the end of transmission of an enquiry message and the beginning of the receipt of a response message, measured at the enquiry terminal.

reusable resource A resource that can be safely used by only one process at a time and is not depleted by that use. Processes obtain reusable resource units that they later release for reuse by other processes. Examples of reusable resources include processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores.

round robin A scheduling algorithm in which processes are activated in a fixed cyclic order; that is, all processes are in a circular queue. A process that cannot proceed because it is waiting for some event (e.g., termination of a child process or an input/output operation) returns control to the scheduler.

scheduling To select jobs or tasks that are to be dispatched. In some operating systems, other units of work, such as input/output operations, may also be scheduled.

secondary memory Memory located outside the computer system itself; that is, it cannot be processed directly by the processor. It must first be copied into main memory. Examples include disk and tape.

segment In virtual memory, a block that has a virtual address. The blocks of a program may be of unequal length, and may even be of dynamically varying lengths.

segmentation The division of a program or application into segments as part of a virtual memory scheme.

semaphore An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. Depending on the exact definition of the semaphore, the decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a *counting semaphore* or a *general semaphore*.

sequential access The capability to enter data into a storage device or a data medium in the same sequence as the data are ordered, or to obtain data in the same order as they were entered.

sequential file A file in which records are ordered according to the values of one or more key fields and processed in the same sequence from the beginning of the file.

server (1) A process that responds to request from clients via messages. (2) In a network, a data station that provides facilities to other stations; for example, a file server, a print server, and a mail server.

session A collection of one or more processes that represents a single interactive user application or operating system function. All keyboard and mouse input is directed to the foreground session, and all output from the foreground session is directed to the display screen.

shell The portion of the operating system that interprets interactive user commands and job control language commands. It functions as an interface between the user and the operating system.

spin lock Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

spooling The use of secondary memory as buffer storage to reduce processing delays when transferring data between peripheral equipment and the processors of a computer.

stack An ordered list in which items are appended to and deleted from the same end of the list, known as the top. That is, the next item appended to the list is put on the top, and the next item to be removed from the list is the item that has been in the list the shortest time. This method is characterized as last in first out.

starvation A condition in which a process is indefinitely delayed because other processes are always given preference.

strong semaphore A semaphore in which all processes waiting on the same semaphore are queued and will eventually proceed in the same order as they executed the wait (P) operations (FIFO order).

swapping A process that interchanges the contents of an area of main storage with the contents of an area in secondary memory.

symmetric multiprocessing (SMP) A form of multiprocessing that allows the operating system to execute on any available processor or on several available processors simultaneously.

synchronous operation An operation that occurs regularly or predictably with respect to the occurrence of a specified event in another process, for example, the calling of an input/output routine that receives control at a precoded location in a computer program.

synchronization Situation in which two or more processes coordinate their activities based on a condition.

system bus A bus used to interconnect major computer components (CPU, memory, I/O).

system mode Same as *kernel mode*.

task Same as *process*.

thrashing A phenomenon in virtual memory schemes, in which the processor spends most of its time swapping pieces rather than executing instructions.

thread A dispatchable unit of work. It includes a processor context (which includes the program counter and stack pointer) and its own data area for a stack (to enable subroutine branching). A thread executes sequentially and is interruptible so the processor can turn to another thread. A process may consist of multiple threads.

thread switch The act of switching processor control from one thread to another within the same process.

time sharing The concurrent use of a device by a number of users.

time slice The maximum amount of time that a process can execute before being interrupted.

time slicing A mode of operation in which two or more processes are assigned quanta of time on the same processor.

trace A sequence of instructions that are executed when a process is running.

translation lookaside buffer (TLB) A high-speed cache used to hold recently referenced page table entries as part of a paged virtual memory scheme. The TLB reduces the frequency of access to main memory to retrieve page table entries.

trap An unprogrammed conditional jump to a specified address that is automatically activated by hardware; the location from which the jump was made is recorded.

trap door Secret undocumented entry point into a program, used to grant access without normal methods of access authentication.

Trojan horse A computer program that appears to have a useful function, but also has a hidden and potentially malicious function that evades security mechanisms, sometimes by exploiting legitimate authorizations of a system entity that invokes the Trojan horse program.

trusted system A computer and operating system that can be verified to implement a given security policy.

user mode The least-privileged mode of execution. Certain regions of main memory and certain machine instructions cannot be used in this mode.

virtual address The address of a storage location in virtual memory.

virtual machine One instance of an operating system along with one or more applications running in an isolated partition within the computer. It enables different operating systems to run in the same computer at the same time as well as prevents applications from interfering with each other.

virtual memory The storage space that may be regarded as addressable main storage by the user of a computer system in which virtual addresses are mapped into real addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.

virus Software that, when executed, tries to replicate itself into other executable code; when it succeeds the code is said to be infected. When the infected code is executed, the virus also executes.

weak semaphore A semaphore in which all processes waiting on the same semaphore proceed in an unspecified order (i.e., the order is unknown or indeterminate).

word An ordered set of bytes or bits that is the normal unit in which information may be stored, transmitted, or operated on within a given computer. Typically, if a processor has a fixed-length instruction set, then the instruction length equals the word length.

working set The working set with parameter Δ for a process at virtual time t, $W(t, \Delta)$ is the set of pages of that process that have been referenced in the last Δ time units. Compare *resident set*.

worm A destructive program that replicates itself throughout a single computer or across a network, both wired and wireless. It can do damage by sheer reproduction, consuming internal disk and memory resources within a single computer or by exhausting network bandwidth. It can also deposit a Trojan that turns a computer into a zombie for spam and other malicious purposes. Very often, the terms "worm" and "virus" are used synonymously; however, worm implies an automatic method for reproducing itself in other computers.