CHAPTER

6 Interprocess Communications

Learning Objectives

After reading this chapter, you should be able to:

- Describe the purpose and elements of interprocess communications.
- Discuss the two models of interprocess communications—shared memory region and message passing.
- Explain the various interprocess communication schemes such as signal system, shared memory, message queue, pipe, and FIFO.

6.1 Introduction

Communication is a means of sharing a piece of information between at least two agents. A communication consists of an individual sending some information and another individual receiving that information. Five elements are involved in a communication: (1) the situation (when, where, and what circumstance) that triggers the communication, (2) the information to be communicated, (3) the sender of the information, (4) the medium of transmitting the information, and (5) the receiver of the information. The first element here is not of interest because the operating system only helps processes exchange information and does not create situations for production of information.

In systems involving multiple processes, concurrent processes may interact with one another for many reasons. Two primary objectives of process interactions are: (1) exchange of information to achieve some common goal, and (2) coordination (i.e., synchronization) of activities of the participating processes. Each interaction involves exchanging a finite amount of information between two or more processes. This exchange is accomplished by transferring data from one process to another. Thus, the two flavours comprising process interactions, namely, interprocess communication

≫ Communication is vital for any activity involving two or more agents. The agents involved in communications in operating systems are processes, threads, kernel paths, and processors. Unless stated otherwise, we use the process as the generic agent in this chapter. Communication can be established in many ways but its purpose is principally one, that is, to transfer information from one process to another. The agents "interact" among themselves through communications to achieve this objective.

(IPC) and process synchronization form the two fundamental concepts in designing multiprocess operating systems. We study IPC and related issues in this chapter, and synchronization and related issues in the next.

6.2 Interprocess Communications

Some applications create more than one process to perform application-specific tasks. These processes work together to collaboratively accomplish the tasks. They are called *cooperating processes*. They exchange information among themselves to perform their tasks. Their interaction is direct. Each process is aware of the presence of the other processes in the system, and its execution flow may depend on the executions of the other processes. In addition, the kernel also communicates with application processes in order to coordinate their activities.

At certain points in its execution, a process may decide to send a piece of data to another process. In such an event, the former process is called the *producer* of the data, and the latter the *consumer* of the data, and the data itself a *message*. Producers generate data and send them in the form of finite-size messages to the consumers. The latter receive these messages and consume the data contained in them. For example, a copy program produces blocks of data that are consumed by a disk server. One instance of exchange of data among two or more processes is called an *interprocess communication*, IPC. IPCs are vital in keeping cooperating processes informed about their collective activities.

In a typical communication between two processes, one process sends a message and the other receives it. In many applications, the roles of these two processes are fixed and the communication is repeated many times. Consider a simple example of two processes with a file containing video frames in an encoded form. The objective is to decode the frames and display them on the screen one by one in a specified order. One process is assigned to repeat the activity of producing decoded frames and the other the activity of displaying the decoded frames on the screen. The two processes are allowed to work concurrently. This type of communication commonly occurs in many large interactive applications. Due to its ubiquitous nature and importance, this pattern of communication is abstracted simply as a producer-consumer problem. Other recurring patterns of communications are abstracted with specific names and will be discussed later in this chapter.

Cooperating processes are "loosely" connected in the sense that they each have an independent private address space and proceed at different speeds. Their relative speed is unknown. One process cannot control the speed of another. From time to time, they exchange information among themselves. As noted in Section 1.5.2 on page 20, one process cannot access elements from the private address space of another. This implies that processes need help from the operating system to set up communication facilities among themselves. Modern operating systems support many IPC schemes for this purpose. For each such scheme, the operating system implements a few

communication primitives (i.e., interface operations). Processes perform local computations in their respective private address spaces, and execute these primitives in the kernel space to facilitate the transfer of data across their address space boundaries.

Many IPC schemes have been proposed in literature for various purposes. The schemes are classified into two broad categories: synchronous and asynchronous.

In a communication scheme where each sender waits until a receiver (or a specific receiver) is ready to receive the data is called a *synchronous communication* scheme. When both are ready for the communication, the data exchange takes place between them. Some authors call it a *handshake-* or *rendezvous* scheme.

In contrast, in an *asynchronous communication* scheme, a sender and a receiver may not handshake to exchange data. The sender does not wait until a receiver (or a specific receiver) is ready to receive the data. Instead, the sender stores the data in some a priori known temporary storage (called a *buffer* or *medium of information transmission*),[1] and the receiver obtains the data from that storage later. The buffer is persistent in the sense that until the receiver retrieves the data, it remains in the buffer. There are two alternatives for informing the receiver about the data.

- After putting the data in the buffer, the sender may interrupt the receiver. This is called pushing.
- Alternatively, the receiver may examine the buffer at regular intervals to see if there is any data intended for it. This is called polling.

In this chapter, we will discuss only a few asynchronous IPC schemes that are used in many modern operating systems. Before presenting those schemes, we present two models of IPC.

6.3 Interprocess Communication Models

Processes exchange information either by sending and receiving messages among themselves, or by reading- and writing data in a shared storage space. Thus, there arise two major IPC models: (1) message passing model, and (2) shared memory model. The semantics of communication primitives for these two models are quite different. Both models are widely used. Both are asynchronous communications schemes.

≫ Shared-memory models and message-passing models are functionally equivalent. That is, a program written for a shared-memory model can be transformed to a message-passing model and vice versa without changing the intent of the program.

6.3.1 Message-passing Model

In the message-passing model, an instance of information exchange involves copying data from one process address space into another via the kernel space (see Fig. 1.15 on page 29). The operating system allocates and then manages

[1] A *buffer* is a piece of storage space that is used to temporarily store data that is being transferred between two agents.

the required buffer space in the kernel to hold unreceived messages. A sender-process deposits a message in the buffer, and a receiver-process retrieves the message from the buffer later. There are many cases of this model. A sender-process may send a message to one particular process, to some processes in a group, or to all processes in a group.

Send and receive are the basic communication primitives in this model and they have many variants to emulate various cases. A send-primitive takes a message as its argument, and helps the sender to copy the message from its private address space into the kernel space. A receive-primitive takes a free block in the receiver's private address space as a parameter and copies a message from the kernel space into the block. Each send- or receive operation involves executing a system call as the data transfer takes place between the user space and the kernel space.

Before exchanging messages, processes need to set up communication buffers, often called channels. A channel is a communication medium in which senders deposit messages and receivers retrieve messages. There are many factors that control the behaviour of a channel such as: (1) Who is the owner of the channel? (2) Can the channel be accessed by multiple processes? (3) Is the channel unidirectional or bidirectional? (For a unidirectional channel, senders and receivers are disjoint.) (4) What is the capacity of the channel? (5) Does the channel handle fixed- or variable-size messages? (6) Does the channel support direct- or indirect addressing? (In direct addressing, the sender specifies a particular receiver to whom it wants to send a message, and a receiver specifies a particular sender from whom it wants to receive a message. In indirect addressing, a message sent can be received by any receiver, and a receiver can receive a message from any sender; a *mailbox* kind of buffer stores all unreceived messages.)

6.3.2 Shared-memory Model

In the shared-memory model, parts of process private address spaces are mapped to the same physical memory (see Fig. 1.16 on page 29) called *shared-memory regions*. The operating system helps processes in setting up and destroying shared-memory regions. Processes can directly write data into and read data from shared-memory regions. Read and write are the basic communication primitives in this model and they do not require making system calls. Processes themselves manage the space in shared regions on their own without intervention from the operating system.

6.3.3 Data Representation

All IPC schemes are modelled as presented in the schematic diagram in Fig. 6.1. Ultimately, all communications involve storing and retrieving information in some storage medium that is shared by all agents involved. To manipulate the available information easily, the shared space is organized in various data structures called shared data or shared variables. In general,

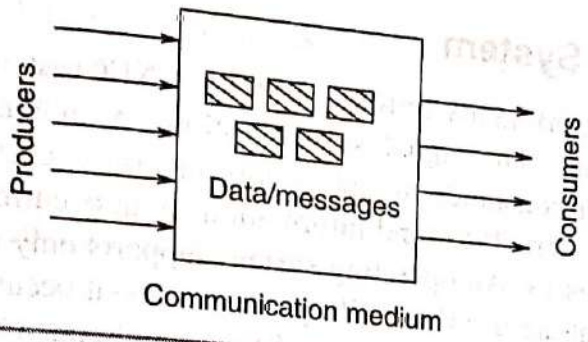


Figure 6.1: *Organizing the medium of information exchange for IPCs.*

a *shared variable* is an abstraction of persistent IPCs. Shared variables are logical units of objects for the purpose of manipulating information by processes. Each shared variable has a unique name or address, and a type. The type defines a finite domain of values, interface operations, and consistency semantics. The variable can store any value from the domain. Interface operations are the only means to access the shared variable. The semantics of the operations describe the permitted behaviour of the operations. Processes communicate among themselves by manipulating values of shared variables by executing operations supported by the variables. A sender process writes new values in the variables that are later read by the receiver processes.

Different IPC schemes use various types of shared variables, and implement different primitives to facilitate IPCs. Senders and receivers invoke these primitives in doing their IPCs. We may need to synchronize accesses to shared variables to ensure integrity of the shared data. (We discuss synchronization in the next chapter.)

If we have an unbounded temporary buffer that can hold an arbitrary number of data items, a producer will never wait to store a data item, but a consumer may need to wait for new data if the buffer is empty. In reality, the buffer is of finite capacity, and can store only a limited number of data items there. Depending on the situation, both consumers and producers may need to wait there. A producer waits only if there is no free space in the buffer, and a consumer waits only if there is no data in the buffer.

6.4 Interprocess Communication Schemes

Various IPC schemes have been proposed in literature (and many of them are implemented in some operating system or other). We discuss here IPC schemes from UNIX systems, and they include signal system, shared memory region, message queue, pipe, FIFO, and socket. One scheme differs from another by the way the temporary storage medium (i.e., buffer) is structured to store information and nature of communication primitives they provide. Except the shared memory region, all other schemes require processes to make system calls to transfer data (even of a single bit) across process address spaces. The above-mentioned six IPC schemes are discussed in the following six subsections.

≫ There are three main sources of signals in UNIX. They are: (1) exception from the execution of current instruction, (2) a user action by an input device, and (3) execution of the kill system call.

6.4.1 Signal System

The signal system, originally introduced in the UNIX system, is the simplest of all IPC schemes. The signal system follows a simpler form of asynchronous producer-consumer model of information exchange. It helps the processes (and the kernel) to send information about occurrences of events to one or more processes. An operating system supports only a limited number of signal types that are used to notify processes about occurrences of a fixed set of predetermined events. Each signal type represents a single event from this set. When an event occurs, a process (or the kernel) notifies its occurrence to a single process or to a group of processes by sending the corresponding signal information to them. For example, the kernel sends one specific signal to a process when one of its children processes exits. A keyboard interrupt can also generate a signal, and the kernel sends it to the process that owns the keyboard.

≫ Unlike other IPC schemes, a process does not need to explicitly set up a signal descriptor by itself; the operating system does so as part of the process creation operation at the time of the process descriptor initialization.

Each process has a reserved space in the kernel (usually, as a part of the process descriptor) to store signal-related information. This space is called the *signal descriptor*. A single bit variable stores information for a specific signal type, as shown in Fig 6.2. The bit pattern in the figure indicates at least two events (of types 1 and 4) occurred and the process is informed of the two events. There are two actions involved for each signal-sending operation, namely delivery and receipt. The former is performed by the kernel, and the latter by the target process. When a specific signal is sent to a process, the signal is considered to have been *delivered* to the process only when the kernel has set the corresponding bit variable in the signal descriptor of the process. When the process receives the signal, the kernel resets the bit variable. The signal subsystem does not record the signal-sender information. A delivered signal may not be received by the target process immediately. There can be at the most one unreceived signal of a given signal type; the other signals of the same type are considered *lost*. Consequently, if the same signal is sent to a process repeatedly without being received by the process, all but the last instance of the signal are considered lost. A signal sent to a process cannot be received by the process more than once.

≫ Signal handlers are procedures for processing (or catching) the signal. The operating system provides handlers to signals that it defines. A process can replace some of these handlers by its own handlers in order to deal with those signals in its customized own way.

A process eventually receives a signal sent to it unless the process terminates. However, if signals of two different types are delivered to a process at the same time, there is no guarantee of the order in which the process will receive the signals. There is no relative priority among signals. On receipt of a signal, the receiving process has one of two possible courses of actions: (1) ignore the signal (and hence the corresponding event), or (2) execute a special function called the *signal handler* to handle the occurrences of the corresponding event. To handle a particular signal in its own way, a process needs

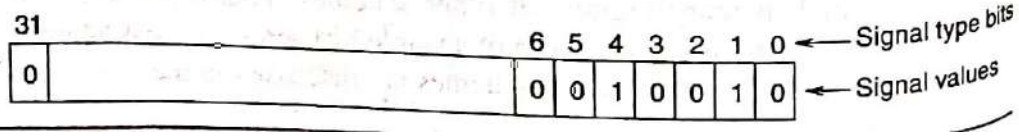


Figure 6.2: *A typical signal descriptor.*

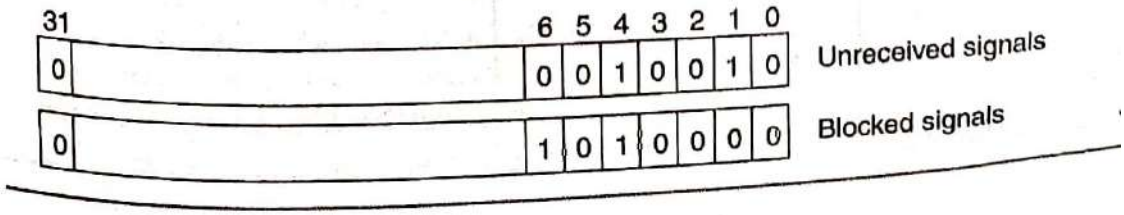
to register a signal handler with the operating system for the said signal type. On receiving the signal, the process stops what it has been doing, and starts executing the signal handler. If a process does not explicitly inform the operating system of one of these two options for a signal and the signal is indeed sent to the process, the operating system performs a default action. Default actions depend on the signal type, and the actions generally are (1) ignore the signal, (2) terminate the process, (3) produce a core dump and terminate the process, (4) suspend the process, and (5) resume the suspended process.

A subtle point to note here is that a process receives signals in the kernel space. But, if there is a registered signal handler, it resides in the user space. The process cannot do a normal return from the kernel space to the user space to execute the signal handler. The kernel makes a kind of upcall in the user space. The process temporarily leaves the kernel space, executes the signal handler, and immediately returns to the kernel space when the handler execution is complete. Note that during the signal handler execution, the process can enter the kernel space by making system calls. (We will revisit Linux way of signal handling in Section 17.6.1.)

A process may temporarily block those signals that are of no interest to it. The process is not interested in these signal types for the present. Note that these signals can be sent to the process, and the kernel does deliver these signals to the process. However, the process will not receive blocked unreceived signals, neither will the kernel perform default actions until the process unblocks the signal types. For every signal type, the operating system maintains another bit variable in the signal descriptor to indicate whether or not the signal type is temporarily blocked. See signal descriptor in Fig. 6.3, signal type-4 is blocked by the process. It will not handle that unreceived signal until it unblocks the signal type.

POSIX standards define about 20 signals. (However, most systems support more than 20 signals. Normally the number of bits available in an integer variable limits the number of signals.) POSIX identifies each signal by a name prefixed with SIG—for example, SIGTERM, SIGSEGV, etc.

Signal sending and handling in single threaded systems is straightforward because all signals are sent and handled by the same thread. In multithreaded systems signals can be handled by different threads in the target process. All threads share the same signal handlers, but they can have different signal masks to block specific signal types. Synchronous signals (those that are generated by the executing thread, for example, division by zero) are delivered to the executing thread. Asynchronous signals come from outside the process, and for such signals we have a choice when the receiving process has multiple threads. Do we deliver an asynchronous signal to any thread, a few



particular threads, or all threads? Some systems allow threads to specify which signals they are willing to handle, and the system delivers those signals to those threads only. As each signal can only be received once by the process, the system may deliver the signal to the first thread that has not blocked the signal. There are other systems that create a default thread that handles all asynchronous signals.

≫ You may recall from Chapter 4 that threads of one process are not visible to another process. Consequently, one process cannot direct a signal to a particular thread in another process.

6.4.2 Shared Memory Region

A signal conveys only the minimum information about the occurrence of a predefined event, that is, the name of the event. Also, every signal communication is explicitly routed through the kernel. The size, content, and control restrictions are relaxed in the shared memory communication mechanism. A *shared memory* is an abstraction of persistent IPC, and it can hold a collection of related shared variables. Shared variables are used to exchange information among the cooperating processes. Each shared variable is accessed by a predefined set of primitive operations. Processes may execute these operations, often concurrently, to read- and write shared variables. The behaviour of operation executions is required to be "consistent" for effective IPCs.

≫ Shared memory is a generic and one of the most efficient IPC mechanisms. Since it can be accessed like any other memory area, programming using shared memory is straightforward.

Processes in general cannot access each other's private address spaces. The operating system prohibits such access violations. It helps processes in mapping parts of their address spaces to the same physical memory locations. The mapping is set up in the abstraction of memory regions. A *memory region* is a logical entity in an address space, which has a base address and a specified length. As shown in Fig. 6.4, regions (of different processes) of the same length can only be shared, that is, mapped to the same (sized) physical memory region. Once a shared memory region becomes a part of process address spaces, the corresponding processes can access the region without causing address violation exceptions.

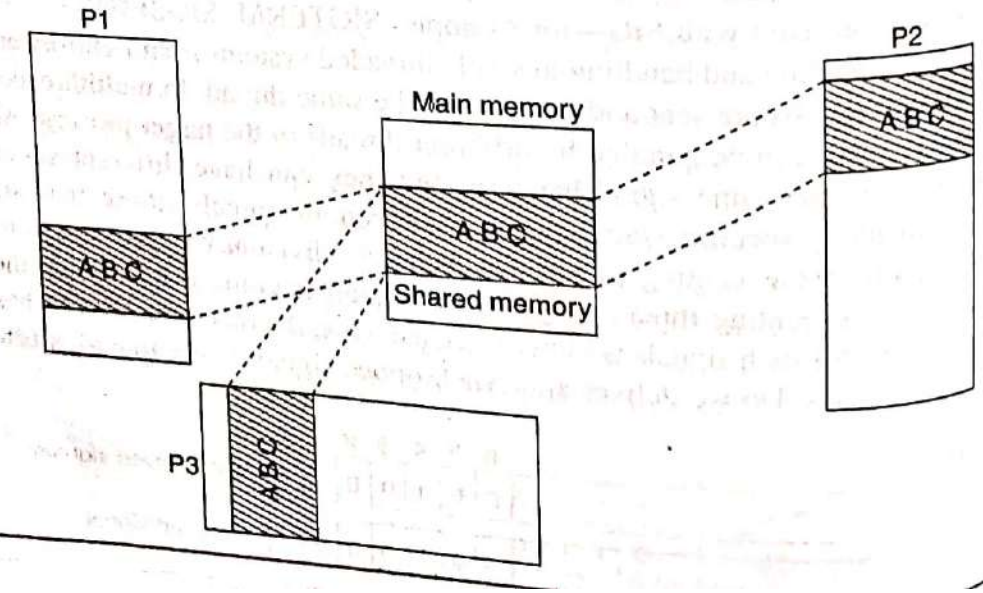


Figure 6.4: *An abstract view of shared memory.*

The operating system implements system call services to create new shared memory regions and to attach existing memory regions to process address spaces. The system associates a unique key with each shared memory region created in the system. Processes use the key to attach (and detach) a shared memory region to (and from) their address spaces. The processes can read from and write into a memory region by executing ordinary memory read and write instructions.

The operating system only creates shared memory regions and attaches them to processes. Granularity of data (i.e., data types) is decided by processes themselves, and not by the operating system. The synchronization of operation executions on shared data is solely done by applications themselves, and not by the operating system. In short, programs for manipulating data in shared memory regions are parts of applications themselves, and not of the operating system. The operating system helps processes in creation, maintenance, and destruction of shared memory regions.

≫ A shared memory region may be mapped to different logical addresses in different processes, see Fig. 6.4. The memory management system translates the different logical addresses to the same physical address pertaining to the shared memory region. We study memory management and runtime address translation schemes in Chapter 8.

6.4.3 Message Queue

A message queue is an abstraction of an asynchronous interprocess message communication scheme with a specified structure and access pattern. A *message queue* is a data structure (usually linked list) that is set up to allow one or more processes to add messages that are removed by other processes. A *message* here is a representation of some structured information, and it is finite in size. Message queues reside in the kernel space, but the operating system is not concerned about message contents. The system does not lose- or alter messages. A process sends a message by enqueuing the message in the message queue. The message is not addressed to a specific process. The message is later dequeued by another process. A message can be dequeued only once, and also, the same message cannot be dequeued by many processes. The message queue service discipline is strictly FIFO.

≫ For other IPC schemes that are discussed in this chapter, the synchronization is provided by the kernel.

The following message queue primitives are supported by some operating systems: (1) creating a new message queue, (2) opening an existing message queue, (3) sending a message to the queue, (4) receiving a message from the queue, (5) closing the queue, and (6) destroying the queue. Before the actual communications begin, a message queue must be created. When a message queue is created, the operating system allocates a finite amount of space in the kernel to temporarily store unreceived messages. The space depends on the size of the message queue, and is referred to as the *message buffer*. Processes open a message queue before exchanging messages through the queue. A producer produces a message and puts it in the message buffer from where a receiver obtains it. The message communication is subject to two resource constraints: (1) producers cannot send more messages than the buffer can hold, and (2) receivers cannot consume messages faster than producers produce them. When a producer attempts to put a message in a full buffer, it may be delayed until a receiver consumes another message from the buffer and makes some space available to the producer. Analogously, when a

≫ A message could be addressed to a receiver process explicitly or implicitly by placing it in a particular queue that the process has access to. In literature, the latter case is called a mailbox and in that sense a UNIX message queue is a mailbox. A message queue simply emulates a bounded size FIFO queue.

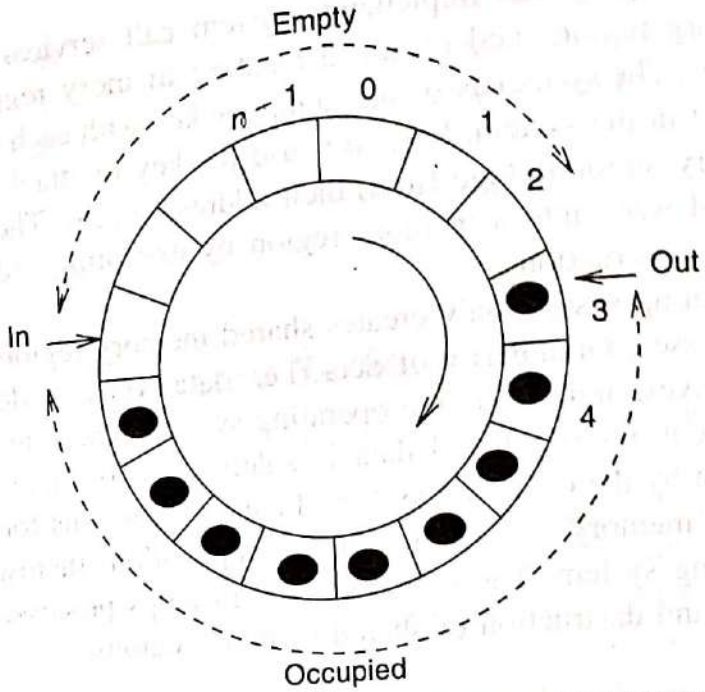


Figure 6.5: *Structure of a bounded message queue.*

receiver attempts to retrieve a message from an empty buffer, the receiver will be delayed until a producer puts a message in the buffer and makes it available to the receiver. To avoid indefinite blocking, a process may opt for the non-blocking mode of operation; when its operation cannot be completed without blocking, it is sent a negative status report instead of blocking it.

In reality, the message buffer of a message queue is treated as a circular storage of a finite number of message entries (see Fig. 6.5). The queue is called a bounded message queue. Figure 6.6 defines **put** and **get** primitives that are executed to send- and receive messages to and from a bounded message queue.[2] (Note that the routines do not take care of any synchronization-related problems; we will discuss them in Chapter 7. For the time being, we assume that processes execute the routines in a mutually exclusive manner and the routine executions are ordered in such a way that no process is blocked in the routines. We will revisit these routines in Section 7.5.9.)

There are n, $n > 0$, entries in the message buffer to hold at the most n unreceived messages. The n entries are manipulated to implement a bounded circular queue. Two integer variables *in* and *out* encompass all entries that contain unreceived messages. The "Occupied" arc in the figure shows this. The shaded ovals are unreceived messages. The integer variable *in* points to the entry where the next message will be put, while *out* points to the entry from where the next message will be received. The integer variable *count* refers to the number of unreceived messages available in the buffer. If *count* is 0, the buffer is empty, and a **get** operation will need to wait. If *count* is n, the buffer is full, and a **put** operation will need to wait. The *count* is incremented by 1 when a new message is put in the buffer, and decremented by 1 when a message is removed from the buffer. A producer produces a new

[2]To conserve print space, we will not present proofs of correctness of any programs in this book. We are primarily interested in learning concepts, and issues involved in designing and developing operating systems rather than finding the best- or absolutely correct solutions to these issues.

```
Constant
   n = maximum number of entries in the message buffer, where n > 0;

Type
   message = structure of buffer entry;

Data structures and initial values
   message buff[n]; /* an array of n message entries */
   int count = 0;      /* number of messages in the buffer; initially empty */
   int in = 0;         /* where the next message will be copied */
   int out = 0;        /* from where the next message will be returned */

void put (message* m)
{
   while (count == n) { wait; }  /* message buffer is full */
   buff[in] = *m;                /* copy message in the buffer */
   in = (in + 1)%n;
   count = count + 1;
}
void get (message *m)
{
   while (count == 0) { wait; }  /* message buffer is empty */
   *m = buff[out];               /* copy message from the buffer */
   out = (out + 1)%n
   count = count - 1;
}
```

Figure 6.6: *A typical implementation of a bounded message queue.*

message, and executes the **put** routine to copy the message in the buffer. A receiver executes the **get** routine to copy the oldest message from the buffer.

6.4.4 Pipe

A *pipe* (also called unnamed pipe) is a "one-way" flow of data between two related processes (see Fig. 6.7). It is a fixed size first-in first-out communication channel, and the size is system-dependent. For each pipe, one process writes data into the pipe, and another process reads the data out of the pipe. A pipe is just like a bounded-message queue. But, unlike a message queue, a pipe is shared only by two related processes, and messages are considered an arbitrary sequence of bytes without any message boundary. Reading from an empty pipe or writing into a full pipe causes a process to be blocked until the state of the pipe changes.

≫ As a pipe is an unnamed object, it cannot be shared by unrelated processes.

In UNIX systems, a pipe is implemented as a sequential file. However, a pipe has no name (i.e., it is anonymous) in the file system. The pipe allows users to funnel the output of one program execution into the input of another program execution, and each program execution sees the pipe as an ordinary (sequential) file. A process creates a pipe by executing the **pipe** system call, and then creates a child process. The two processes can communicate through the pipe. The **pipe** system call actually creates two file descriptors for the calling process using which the pipe is accessed. The system call also allocates space to hold unreceived data. Any process that has access to those file descriptors can communicate through the pipe. (When a parent creates a

≫ In UNIX systems, pipes are normally used as unidirectional byte streams which connect the standard output from one process to the standard input of another process. Neither process is aware of this redirection. They behave just as they would normally do in reading and writing their standard input and output.

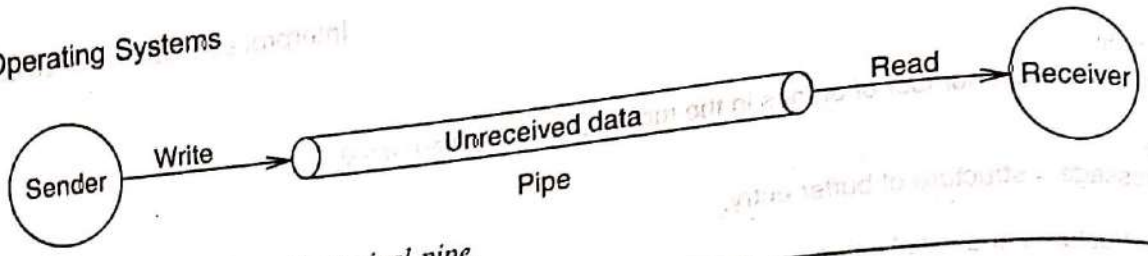


Figure 6.7: *An abstract view of a typical pipe.*

child, UNIX systems automatically duplicate file descriptors of the parent for the child. Thus, the child can access the pipe.) The pipe is closed when all processes close those descriptors. The operating system releases the pipe space. Though not shown in Fig. 6.7, in UNIX a process can both write to and read from the same pipe.

≫ In UNIX, the symbol | is used for pipe in command line. For example, the pipe between two popular commands ls | more takes the output of ls in one side of the pipe and pumps out as the input to more, so producing a paged listing of the current directory. More than two commands can be connected by pipes, one after another.

6.4.5 FIFO

In UNIX systems, a FIFO is a named pipe, operating on the basis of first-in first-out order. A FIFO is actually a special file that is similar to a pipe, except that it is created differently. Instead of being an anonymous communication channel, a FIFO is created in a file system by invoking mkfifo system call. A FIFO is an empty file.

Once a FIFO is created, any process can open it for reading or writing, in the same way as for an ordinary pipe. However, it has to be open at both ends before one can proceed to perform any input- or output operations on it. Opening a FIFO for reading normally blocks it until some other process opens the same FIFO for writing, and vice versa.

≫ A FIFO offers the same communication facilities as a pipe and has a name in the file system so that it can be accessed like a file. This allows the named pipe to be used between unrelated processes.

6.4.6 Socket

A socket is somewhat similar to a pipe, but it is a "bidirectional" first-in first-out communication facility. The socket abstraction of IPCs was originally

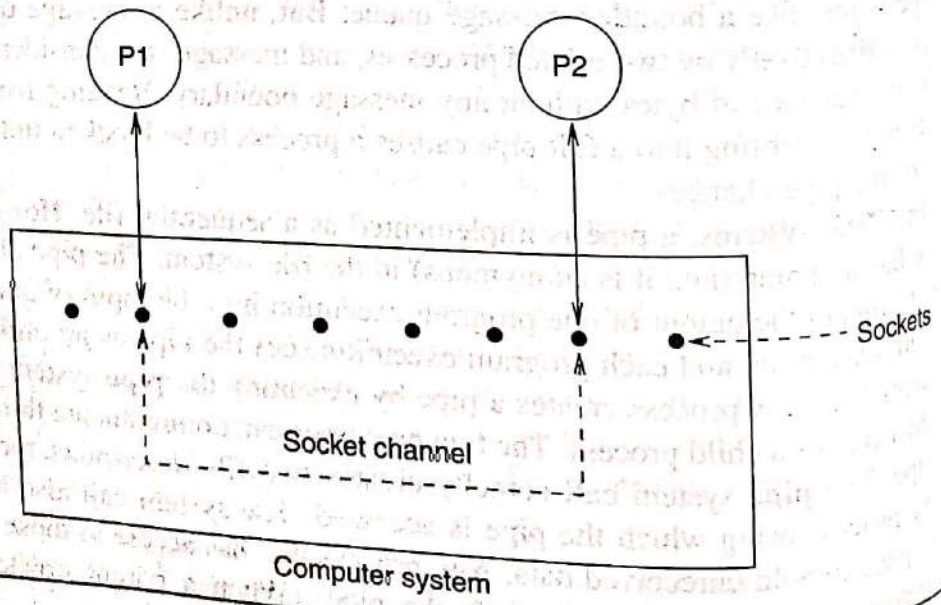


Figure 6.8: *A socket connection between two processes P1 and P2.*

introduced to enable message communications between two processes on two different computer systems, but, as shown in Fig. 6.8, sockets can also be used for communications between processes in the same computer system. A socket is an indirect IPC scheme. Sockets are parts of the communication system proper; each communication channel has two sockets, one at each end of the channel. A process connects to a socket to send and receive data from the socket. Knowing what process is at the other end of the channel may not interest the process. The processes exchange data over the channel by executing read- and write operations on the socket, the same way they access files. We will discuss more about socket in Section 16.5.2, where we talk about IPCs in distributed systems.

≫ IPC sockets are much like electrical sockets. Just as we connect any electric appliance to any electric socket to draw power from the socket, we can connect any process to any IPC socket to exchange data over the socket.

Summary

This chapter discusses the purpose and elements of IPCs. There are two models of IPCs, message passing and shared memory. The chapter presents six IPC schemes from UNIX systems: signal system, shared memory region, message queue, pipe, FIFO, and socket. All these IPC schemes are widely used by application developers.

All schemes except the shared-memory region belong to the message-passing model. They are asynchronous communication schemes. The signal system is the simplest of all. Each signal communication exchanges one bit of information. An application process can send signals to other application processes. The kernel can also send signals to application processes. In fact, in some operating systems, the kernel communicates exception events to processes by sending them signals. The kernel, when it creates an application process, creates a signal descriptor for the process to enable its communication via signals. A process can also block specific signal types, in which case the process is immune to those specific signals. Signal handling in multithread systems is a little problematic. Synchronous signals are always handled by the producing threads. However, asynchronous signals are handled differently in different systems. Some systems create a dedicated thread to handle all asynchronous signals.

For other communication schemes, processes need to set up explicit communication channels. A message queue is a kind of mailbox. All the participating processes have to subscribe to a common message queue. The size of the messages that the message queue can support is arbitrary (subject to some maximum limit). Message **send** and **receive** primitives are atomic, and message queues ensure first-in first-out semantics: messages enqueued first are dequeued first. This chapter presents a circular queue based implementation of message queue. Pipes implement unidirectional flow of data from one process to another "related" process. For each pipe, the operating system reserves a fixed amount of space in the kernel. When a process writes data into the pipe, the kernel copies the data into the reserve space from where another process can read the data out later. FIFOs are named pipes that can be used by any process. They exhibit the same behaviour that pipes do. A socket is somewhat similar to a pipe, but is a bidirectional first-in first-out communication scheme. One socket can only be associated with one process at a time. To set up a communication between two processes, each of them needs to have its individual socket and the kernel set up the communication channel between the two sockets. Each can send- and receive data to and out of the channel through its own socket.

These five types of communications are performed through the kernel space, and synchronization-related issues are taken care of by the kernel. In contrast, in the shared-memory scheme, processes can have shared memory regions that reside in the private address spaces of processes. However, the processes need to create and attach shared memory regions to their address spaces with the help of the kernel. Once a shared memory region is