**3.4 Inter-process Communication**

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process. There are several reasons for providing an environment that allows process cooperation:

* **Information sharing.** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
* **Computation speedup.** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
* **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
* **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an inter process communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of inter process communication: (1) shared memory and (2) message passing.

In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.13.



**Figure 3.13** Communications models.

(a) Message passing.

(b) Shared memory.

Both of the models just discussed are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for inter-computer communication. Shared memory allows maximum speed and convenience of communication. Shared memory is faster than message passing, as message passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In contrast, in shared memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

**3.4.1 Shared-Memory Systems**

Inter-process communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared memory segment. Other processes that wish to communicate using this shared memory segment must attach it to their address space. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let's consider the producer-consumer problem, which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader. The producer-consumer problem also provides a useful metaphor for the client-server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a Web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client Web browser requesting the resource.

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. Two types of buffers can be used. The un bounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full. The following variables reside in a region of memory shared by the producer and consumer processes:

#define BUFFER\_SIZE 10

typedef struct

}item;

item buffer[BUFFER\_SIZE];

int in = 0;

int out = 0;

The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in== out; the buffer is full when ((in+ 1)% BUFFER\_SIZE) == out.

The code for the producer and consumer processes is shown in Figures 3.14 and 3.15, respectively. The producer process has a local variable nextProduced in which the new item to be produced is stored. The consumer process has a local variable next Consumed in which the item to be consumed is stored. This scheme allows at most BUFFER\_SIZE - 1 items in the buffer at the same time.

item nextProduced;

while (true) {

}

*I\** produce an item in nextProduced *\*I*

while ( ((in + 1) % BUFFER\_SIZE) == out)

; *I\** do nothing *\*I*

buffer[in] = nextProduced;

in = (in + 1) % BUFFER\_SIZE;

**Figure 3.14** The producer process.

item next Consumed;

while (true) {

}

while (in == out)

; *II* do nothing

nextConsumed = buffer[out];

out = (out + 1) % BUFFER\_SIZE;

*I\** consume the item in nextConsumed *\*I*

**Figure 3.15** The consumer process.

One issue this illustration does not address concerns the situation in which both the producer process and the consumer process attempt to access the shared buffer concurrently. In Chapter 6, we discuss how synchronization among cooperating processes can be implemented effectively in a shared memory environment.