Processes
(Interprocess Communication)

Contents of Lecture:

- ❖ Interprocess Communication
- ❖ Shared-Memory Systems
- ❖ Message-Passing Systems

References for This Lecture:

- ✓ *Abraham Silberschatz, Peter Bear Galvin and Greg Gagne, Operating System Concepts, 9th Edition,* ***Chapter 3***

Interprocess 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:
  1. **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.
  2. **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 cores.

  3. **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
  4. **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

❖ **Cooperating processes** require an **interprocess communication** (**IPC**) mechanism that will allow them to exchange data and information.

❖ There are two fundamental **models** of **interprocess communication:**
  - ✓ Shared memory
  - ✓ Message passing

❖ **<u>In the shared-memory model:</u>**
  - ✓ 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.

❖ **<u>In the message-passing model:</u>**
  - ✓ Communication takes place by means of messages exchanged between the cooperating processes.

❖ Both of the models just mentioned are common in operating systems, and many systems implement both.

❖ The two communications models are contrasted in next figure (Figure 3.12).

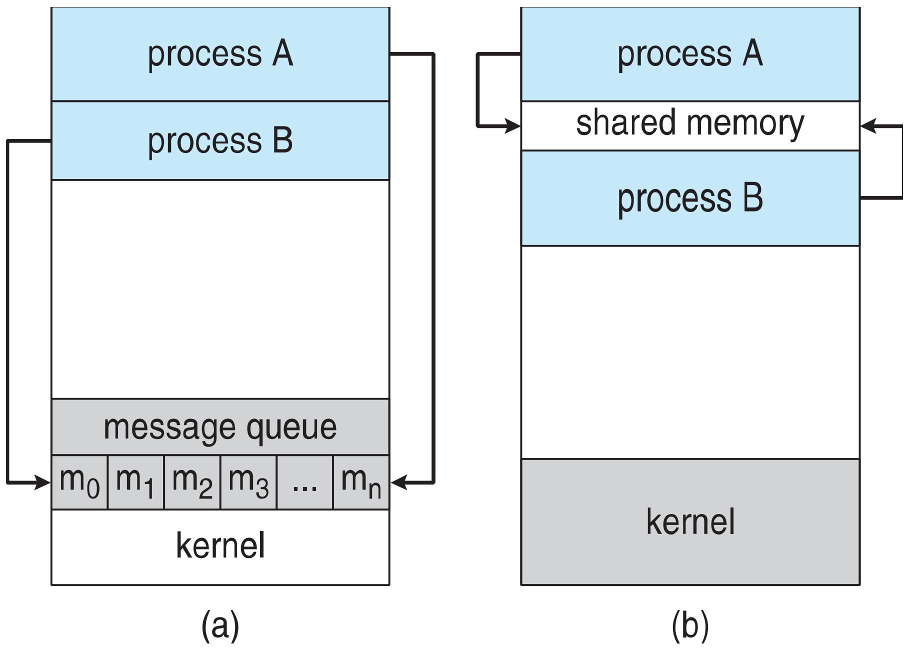

Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

Shared-Memory Systems

- Interprocess 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.

producer–consumer problem

- 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.

- **Examples of producer–consumer problem**
  - ✓ A compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that 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.

Solution to the producer–consumer problem

- 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.

❖ <u>**Two types of buffers can be used:**</u>

- ✓ The **unbounded 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.

❖ Let's look more closely at how the bounded buffer illustrates interprocess communication using shared memory. 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;
```

Where:

- ✓ 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 process is shown in next figure (Figure 3.13).

```
item next_produced;

while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Figure 3.13 The producer process using shared memory.

❖ The producer process has a local variable **next_produced** in which the new item to be produced is stored.

❖ The code for the consumer process is shown in next figure (Figure 3.14).

```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

Figure 3.14 The consumer process using shared memory.

❖ The consumer process has a local variable **next_consumed** in which the item to be consumed is stored

Message-Passing Systems

❖ Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

❖ It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

- ✓ For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

❖ A message-passing facility provides at least two operations:

send(message)
receive(message)

❖ Messages sent by a process can be either fixed or variable in size.

❖ If processes P and Q want to communicate:

- ✓ They must Exchange messages via send/receive operations from each other.
- ✓ They need to establish a **communication link** must exist between them. This link can be implemented in a variety of ways:
  - **Physical implementation** such as:
    - shared memory
    - hardware bus
    - network
  - **Logical implementation**:
    - Direct or indirect communication
    - Synchronous or asynchronous communication
    - Automatic or explicit buffering

We look at issues related to each of these features next.

Related issues

1. Naming

- ❖ Processes that want to communicate must have a way to refer to each other.
- ❖ They can use either **direct** or **indirect** communication.

Direct communication:

- ❖ Each process that wants to communicate must explicitly name the recipient or sender of the communication.

- ❖ In this scheme, the send() and receive() primitives are defined as:
  - ✓ send(P, message) — Send a message to process P.
  - ✓ receive(Q, message) — Receive a message from process Q.

- ❖ A communication link in this scheme has the following properties:
  - ✓ A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
  - ✓ A link is associated with exactly two processes.
  - ✓ Between each pair of processes, there exists exactly one link.

Indirect communication:

- ❖ The messages are sent to and received from mailboxes, or ports. Each mailbox has a unique identification.

- ❖ In this scheme, two processes can communicate only if they have a shared mailbox. The send() and receive() primitives are defined as follows:
  - ✓ send(A, message)—Send a message to mailbox A.
  - ✓ receive(A, message)—Receive a message from mailbox A.

- ❖ A communication link has the following properties:
  - ✓ A link is established between a pair of processes only if both members of the pair have a shared mailbox.
  - ✓ A link may be associated with more than two processes.
  - ✓ Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

2. Synchronization

- Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either **blocking (synchronous)** or **nonblocking (asynchronous)** .

  - ✓ **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
  - ✓ **Nonblocking send:** The sending process sends the message and resumes operation.
  - ✓ **Blocking receive:** The receiver blocks until a message is available.
  - ✓ **Nonblocking receive:** The receiver retrieves either a valid message or a null.

- Different combinations of send() and receive() are possible.
- When both send() and receive() are blocking, we have a **rendezvous** between the sender and the receiver.

- The solution to the producer–consumer problem becomes trivial when we use blocking send() and receive() statements. The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive(), it blocks until a message is available. This is illustrated in next figures (Figures 3.15 and 3.16).

```
message next_produced;

while (true) {
     /* produce an item in next_produced */

     send(next_produced);
}
```

Figure 3.15 The producer process using message passing.

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

Figure 3.16 The consumer process using message passing.