SNS COLLEGE OF TECHNOLOGY

Coimbatore-35.

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

COURSE NAME : 19CSB201 – OPERATING SYSTEMS

II YEAR/ IV SEMESTER

UNIT – II Overview and Process Management

Topic: Process Concept : Inter Process Communication

Mrs. M. Lavanya
Assistant Professor
Department of Computer Science and Engineering

Interprocess Communication

A process can be of 2 types

- 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.

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

Reasons for providing 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 cores.
- **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, listening to music, and compiling in parallel.

Models of IPC

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** and **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.

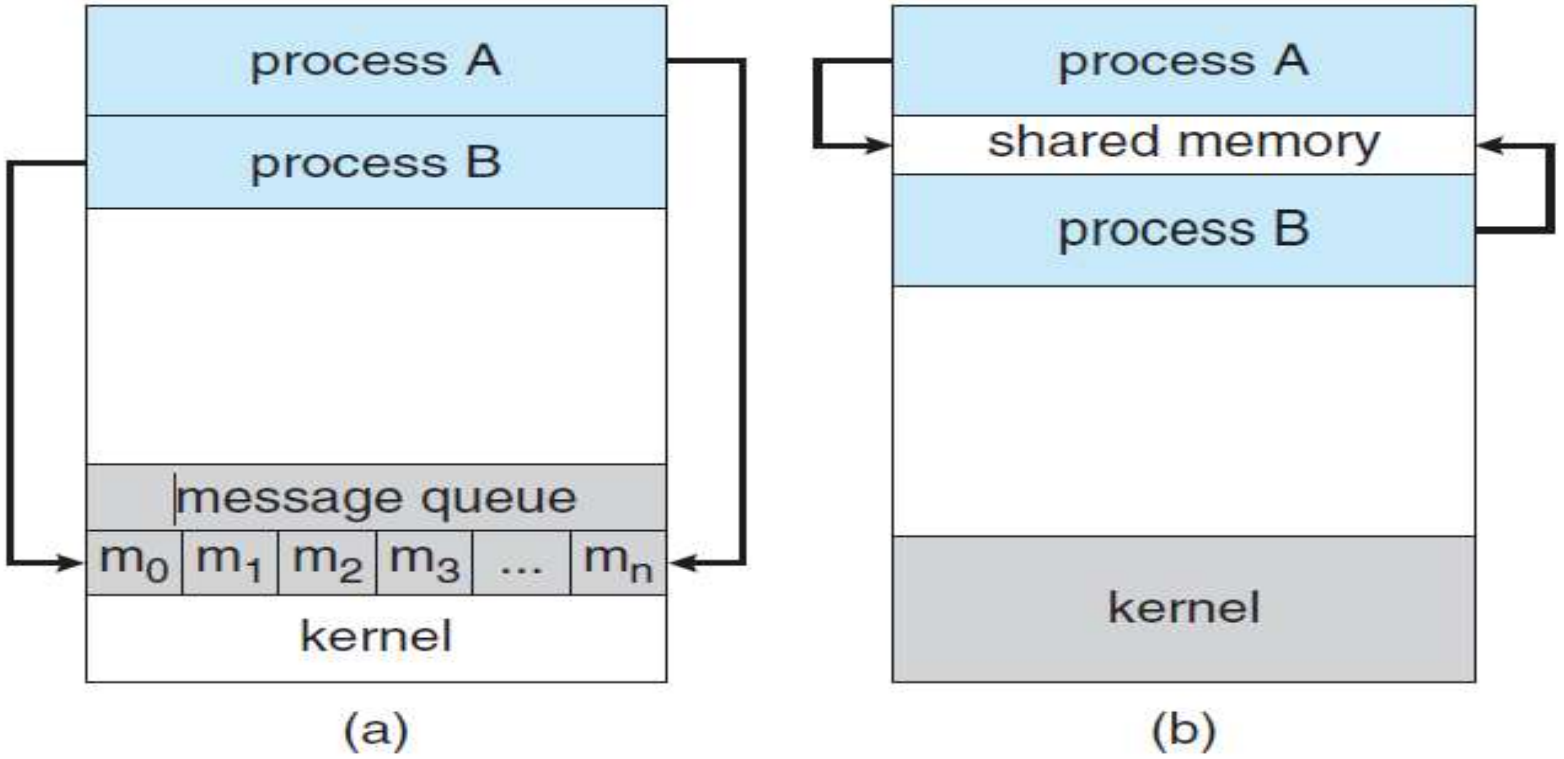

Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.

Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls. Shared memory suffers from cache coherency issues, which arise because shared data migrate among the several caches.

Shared-Memory Systems

Establish a region of shared memory

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.

Example

producer–consumer problem

- **producer** process produces information that is consumed by a **consumer** process. The producer and consumer must be synchronized.

```
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.

```
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 producer–consumer problem uses shared memory

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.

Two types of buffers can be used.

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.

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;
```

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.

A message-passing facility provides at least two operations:

- send(message)
- receive(message)

Methods for logically implementing a link and the send()/receive() operations:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

Naming

Direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication.

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.

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.

This scheme exhibits ***symmetry*** in addressing; that is, both the sender process and the receiver process must name the other to communicate.

A variant of this scheme employs ***asymmetry*** in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender.

In this scheme, the send() and receive() primitives are defined as follows:

- send(P, message)—Send a message to process P.
- receive(id, message)—Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

Indirect communication, the messages are sent to and received from ***mailboxes***, or ***ports***. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

Each mailbox has a unique identification.

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.

In this scheme, 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.

A mailbox may be owned either by a process or by the operating system.

The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

Synchronization

Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous**.

Different combinations of send() and receive() are possible.

- **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 amessage is available.
- **Nonblocking receive**. The receiver retrieves either a valid message or a null.

```
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.

Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.

Basically, such queues can be implemented in three ways:

- **Zero capacity**
- **Bounded capacity**
- **Unbounded capacity**

- **Zero capacity**. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity**. The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity**. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering. The other cases are referred to as systems with automatic buffering.

REFERENCES

TEXT BOOKS:

T1 Silberschatz, Galvin, and Gagne, “Operating System Concepts”, Ninth Edition, Wiley India Pvt Ltd, 2009.)

T2. Andrew S. Tanenbaum, “Modern Operating Systems”, Fourth Edition, Pearson Education, 2010

REFERENCES:

R1 Gary Nutt, “Operating Systems”, Third Edition, Pearson Education, 2004.

R2 Harvey M. Deitel, “Operating Systems”, Third Edition, Pearson Education, 2004.

R3 Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, “Operating System Concepts”, 9th Edition, John Wiley and Sons Inc., 2012.

R4. William Stallings, “Operating Systems – Internals and Design Principles”, 7th Edition, Prentice Hall, 2011


THANK
YOU!