Operating Systems

Instructor: Dr. Shachi Sharma

(Semester: Winter 2019)

Inter Process Communication

Inter Process Communication

- Cooperating Processes
- Independent Processes

Models of Inter Process Communication

- Shared Memory
  - Large amount of data
  - Faster
- Messaging
  - Small amount of data
  - Distributed systems
  - Multi-core systems
  - Usually implemented via system calls

Models of Inter Process Communication

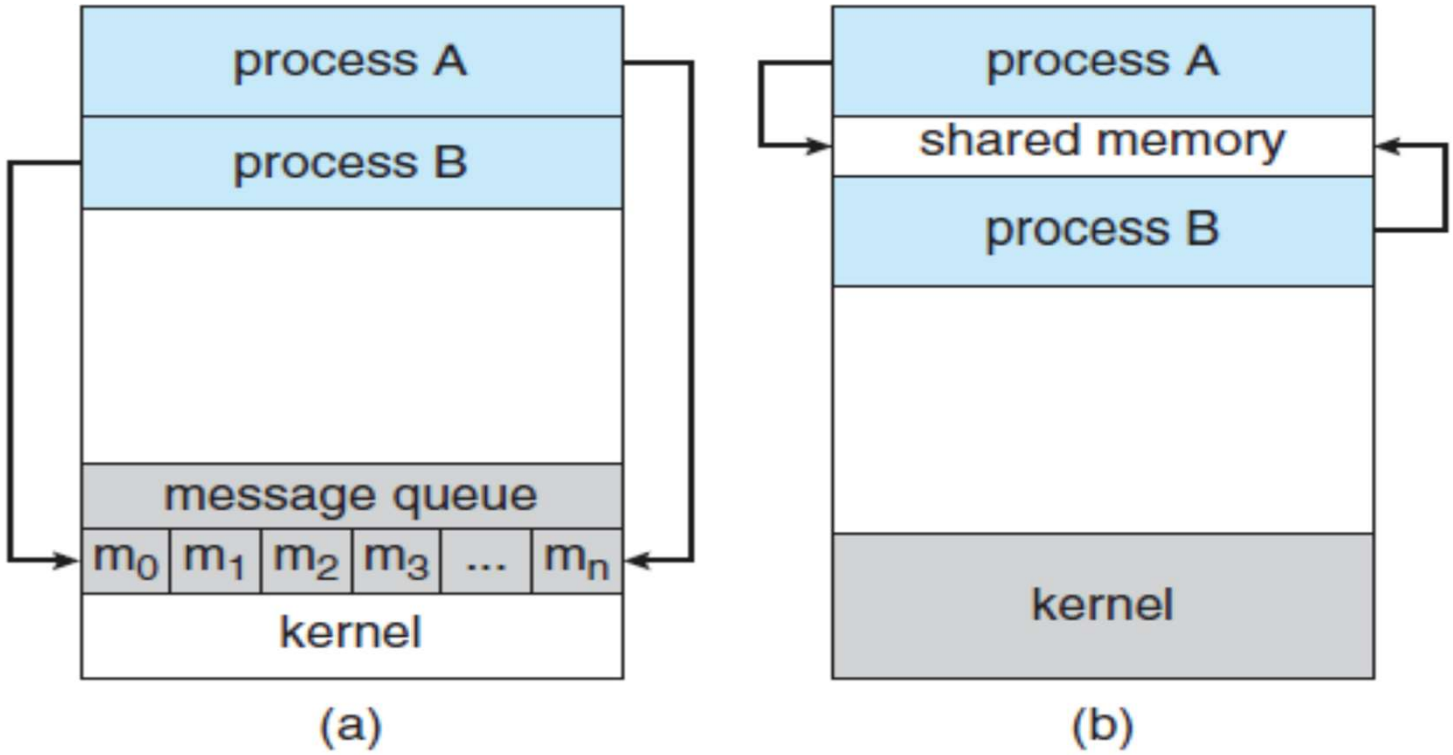

Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

Shared Memory Systems

- Shared memory requires processes to share a region of memory between them
- Typically, shared memory resides in the address space of the process creating it and other communicating processes attach to it
- The communicating processes must ensue that they do not simultaneously write in the shared memory

Producer-Consumer Problem

- A **producer** process produces information that is consumed by a **consumer** process. For example,
  - 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.

Producer-Consumer Problem

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- 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.

Producer-Consumer Problem

```
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;
}
```

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

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

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

Message Passing Systems

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

Message Passing Systems

- If processes *P* and *Q* want to communicate, they must send messages to and receive messages from each other: a ***communication link*** must exist between them. This link can be implemented in a variety of ways
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

Direct Communication - Symmetry

- 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.

Direct Communication-Asymmetry

- 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.
- For example, POSIX message queues use an integer value to identify a mailbox.
- A process can communicate with another process via a number of different mailboxes, but 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.

Indirect Communication

- 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.

Mail Box

- Process owned mail box
- OS mail box
  - Create a new mailbox.
  - Send and receive messages through the mailbox.
  - Delete a mailbox.

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**. 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.