Lecture 5
Process
Management: Process (Continued)

Operations on Processes
1) Process Creation

- During the course of execution, a process may create several new processes. The creating process is called a parent process, and the new processes are called the children of that process.
- Each of these new processes may in turn create other processes, forming a **tree of processes.**
- Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier (or pid),** which is typically an integer number.

In general, when a process creates a child process, that child process will need certain resources to accomplish its task. A child process may be able to obtain its resources from:

1. the operating system, or
2. it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. **Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.**

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated**.**

2) Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.

At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process are deallocated by the operating system.

- Termination can occur in other circumstances as well.
- A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess() inWindows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:

1. The child has exceeded its usage of some of the resources that it has been allocated.
2. The task assigned to the child is no longer required.
3. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

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.

Cooperating processes require an **interprocess communication (IPC) mechanism** that will allow them to exchange data and information. There are two fundamental models of inter-process 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.

The two communications models are contrasted in Figure 3.12.

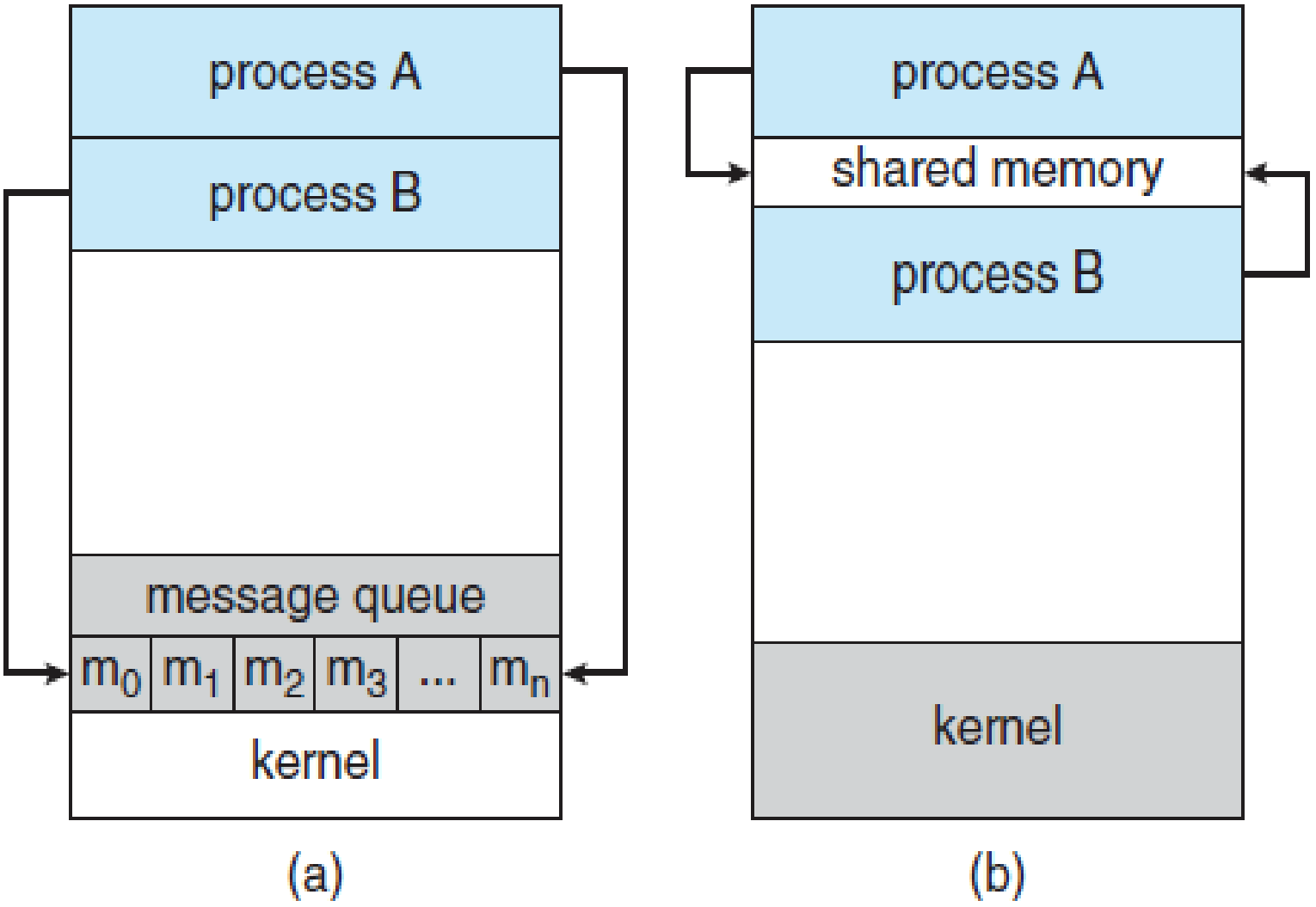

Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

Shared-Memory Systems

The communicated processes can 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 that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.

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)

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. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network) but rather with its logical implementation. Here are several methods for logically implementing a link and the send()/receive() operations:

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

Naming

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

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.

Suppose that processes P1, P2, and P3 all share mailbox A. Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1? The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.
- Allow at most one process at a time to execute a receive() operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message).

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 or nonblocking**—also known as **synchronous and 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.

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, 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.