INTERPROCESS COMMUNICATION (IPC)

- Processes executing concurrently in OS 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 4 reasons for providing an environment that allows process cooperation:
  1. **Information sharing**. Since several users may be interested in the same piece of 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. (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** 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.
- Both of the models are common in OS, and many systems implement both.
- Message passing is useful for exchanging smaller amounts of data.
- Message passing is also easier to implement in a distributed system than shared memory.
- Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
- 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.

- Shared memory suffers from cache coherency issues, which arise because shared data migrate among the several caches.
- As the number of processing cores on systems increases, it is possible that we will see message passing as the preferred mechanism for IPC.

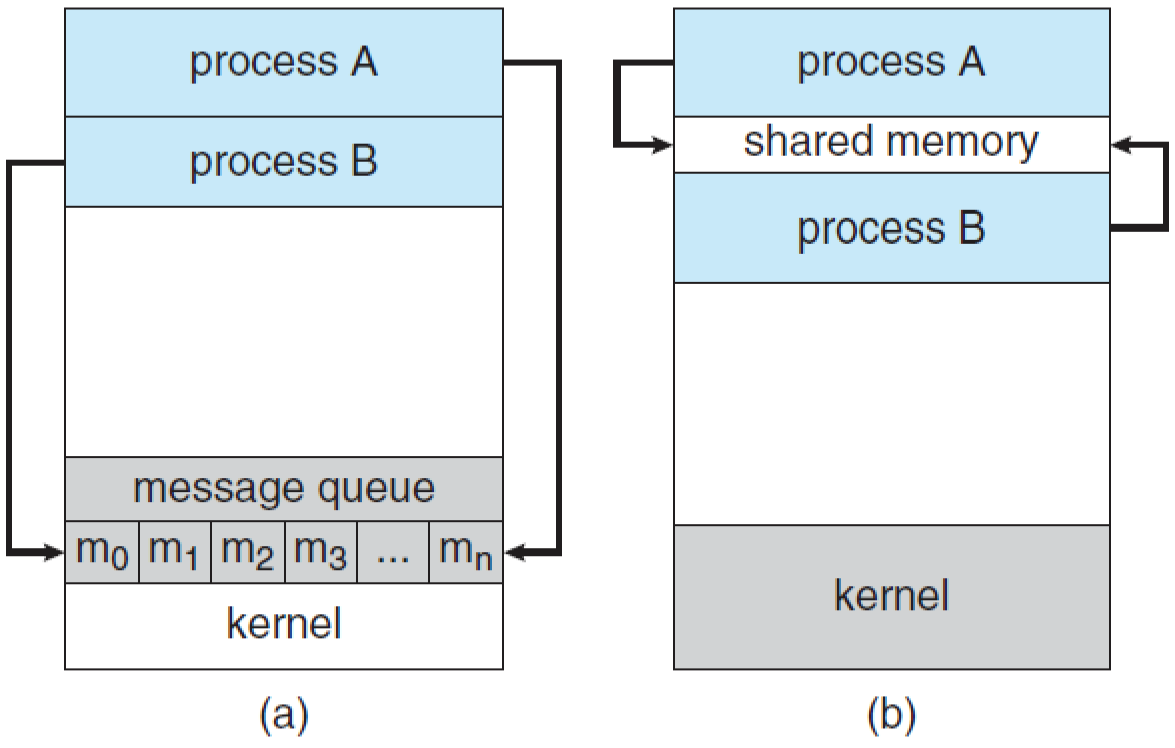

Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

SHARED-MEMORY SYSTEMS

- IPC using shared memory requires communicating processes to 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.
- Normally, OS 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 control of OS.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- Producer–consumer problem is the best example for shared memory.
- A **producer** process produces information that is consumed by a **consumer** process. Both uses the same data buffer.

MESSAGE-PASSING SYSTEMS

- A message-passing facility provides at least two operations: **send(message)** and **receive(message)**
- Messages sent by a process can be either **fixed or variable in size.**
- If only fixed-sized messages can be sent, the system-level implementation is straightforward.

- Variable-sized messages require a more complex system level implementation
- 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 is a logical link rather than physical link.
- Here are several methods for logically implementing a link and the send()/receive() operations:
  - Direct or indirect communication (Naming)
  - Synchronous or asynchronous communication (Synchronization)
  - Automatic or explicit buffering (Buffering)

Naming

- Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication.
- 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 has the following properties:
  - A link is established automatically between every pair of processes that want 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 disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity. Changing the identifier of a process may necessitate examining all other process definitions.
- All references to the old identifier must be found, so that they can be modified to the new identifier.

- With **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 (usually an integer).

- 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.
- 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 OS.
- If the mailbox is owned by a process, then the mailbox will be a part of the address space of that process
- Owner can only receive messages through this mailbox and the user can only send messages to the mailbox.
- When a process that owns a mailbox terminates, the mailbox disappears.
- Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

- In contrast, a mailbox that is owned by the OS has an existence of its own.
- It is independent and is not attached to any particular process.
- The OS 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.
- The process that creates a new mailbox is that mailbox's owner by default.
- Initially, the owner is the only process that can receive messages through this mailbox.
- However, the ownership and receiving privilege may be passed to other processes through appropriate system calls.

Synchronization

- 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.
- 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.

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:
  1. **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.
  2. **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.
  3. **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.