Processes executing concurrently in the OS may be either independent (doesn't share data with any other processes executing in the system) or cooperating processes (it can affect by the other processes executing in the system).

Reasons to provide environment that allows process cooperation:

- Information sharing. Since several apps may be interested in the same piece of information (for example, copying and pasting) we must provide an environment to allow concurrent access to such information
- Computation speedup. We can break particular task into subtasks, each of which will be executing in parallel with the others. Such speedup can be achived only if the computer has multiple cores
- Modularity. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data. There a two fundamental models: shared memory and message passing.

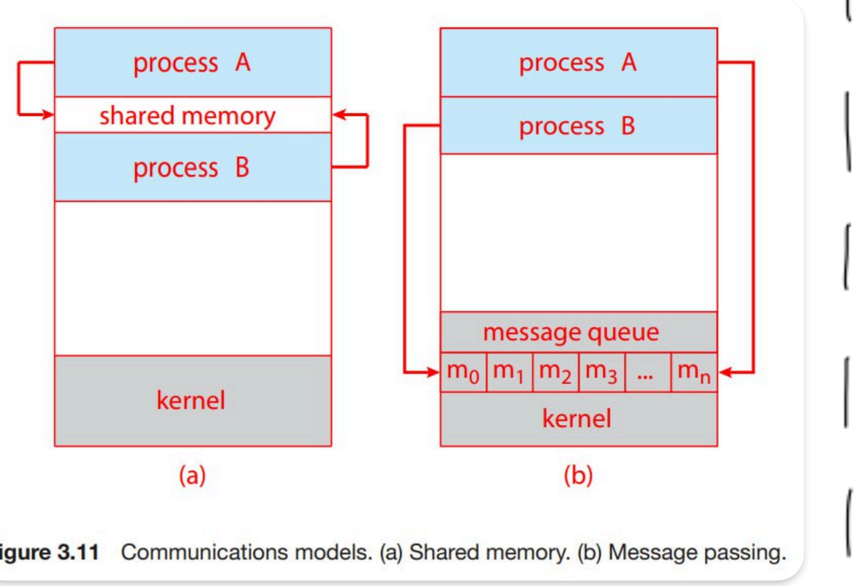

Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

In the shared memory model, a region of memory that is shared by the cooperating processes is established. Processes can 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.

Message-passing is useful for exchanging smaller amounts of data, because no conflicts need to be avoided. Shared memory can be faster, since message-passing systems are typically implemented using system calls, and system calls are required only to establish shared-memory region that resides in the address space of the proc creating the s-m.

To illustrate the concept of cooperating processes, consider the producer-consumer problem.

One solution uses shared memory. To allow producer and consumer processes to run concurrently, we must have avaible 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 both processes. Both processes must be synchronized so that consumer doesn't try to consume an item that has not yet been produced.

Two types of buffer can be used:

- Unbounded buffer places no practical limit on the size of buffer. The consumer may have to wait for new items, but producer can always produce new items
- Bounded buffer assumes a fixed buffer size. In this case consumer must wait if the buffer is empty and producer must wait if the buffer is full

Another solution for cooperating both producer and consumer via message-passing facility. It provides a mechanism to allow both processes communicate and synchronize their actions without sharing the same address space. It useful for distributed systems.

A msg-passing facility provides at least two operations:

- send(message) and receive(message)

Message sent by a process can be either fixed or variable size. The fixed size messages makes the system-level implementation is straightforward. but makes the task of programming more difficult. Conversly with var-size msg.

If a process P and Q want to communicate, they must send message to and receive from each other: a communication link must exist between them. 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. Under direct communication, process must explicitly name the recipient or sender of the communication:

- send(P, msg) – send message to process P
- receive(Q, msg) – receive 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 – both, the sender and the receiver processes must name the other to communicate. A variant of this scheme employs asymmetry in addressing. Here, only sender names the recipient; the recipient is not required to name the sender:

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

The disadvantage in both of these schemes is the limited modularity of the resulting process definitions. Any such hard-coding techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described beat.

With indirect communication, the messages are sent to and received from ports, or mailboxes – object into which messages can be placed by process and from which messages can be removed. Each mailbox has a unique identification. 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.

- send(A, msg) – Send a message to mailbox A.
- receive(A, msg) – 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.

If more than two processes (P_1, P_2, P_3) share mailbox, and P_1 sends a message to A there are the following methods for receiving processes:

- Allow a link to be associated with two process at most.
- Allow at most one process at a time to execute a receive() operation.
- Allow the system to select arbitraliny which process will receive the message The system may define an algorithm for selecting which process will receive the message. The system may identify the receiver to the sender

A mailbox may be owned either by a process or by the OS. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner, which can only receive messages through this mailbox, and the user (which can only send messages to the mailbox. Since each mailbox has a unique owner, there can be no confusion about which process receive a message. When a process that owns a mailbox terminates, the mailbox disappears.

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 following:

- Create a new mailbox and delete it
- Send and receive messages through the mailbox

The process that creates a new mailbox is that mailbox's owner by default However, the ownership and receiving privelege may be passed to other process through appropriate system call.

Message-passing may be blocking or nonblocking (sync or async).

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

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Such queues can be implemented in three ways:

- Zero capacity. The link cannot have any messages waiting in queue. The sender must block until the recipient receives the message.
- Bounded capacity. At most n messages can reside in it. If queue isn't full the sender can continue execution without waiting. If the queue is full, the sender must block until space is avaible in the queue.
- Unbounded capacity. The sender never blocks.

A pipe acts as conduit allowing 2 processes to communicate. Pipes properties:

- Unidirectional or bidirectional communication
- If 2-way communication is allowed, full duplex (data can travel in both directions at the same time) or is it half duplex (data can travel only one way at a time)
- Must a relationship exist between communicating processes? (such as parent-child)
- Can the pipes communicate over network?

- A process is a program in execution, and the status of the current activity of a process is represented by the program counter, as well as other registers.
- The layout of a process in memory is represented by four different sections: (1) text, (2) data, (3) heap, and (4) stack.
- As a process executes, it changes state. There are four general states of a process: (1) ready, (2) running, (3) waiting, and (4) terminated.
- A process control block (PCB) is the kernel data structure that represents a process in an operating system.
- The role of the process scheduler is to select an available process to run on a CPU.
- An operating system performs a context switch when it switches from running one process to running another.

- The fork() and CreateProcess() system calls are used to create processes on UNIX and Windows systems, respectively.
- When shared memory is used for communication between processes, two (or more) processes share the same region of memory. POSIX provides an API for shared memory.
- Two processes may communicate by exchanging messages with one another using message passing. The Mach operating system uses message passing as its primary form of interprocess communication. Windows provides a form of message passing as well.
- A pipe provides a conduit for two processes to communicate. There are two forms of pipes, ordinary and named. Ordinary pipes are designed for communication between processes that have a parent–child relationship. Named pipes are more general and allow several processes to communicate.
- UNIX systems provide ordinary pipes through the pipe() system call. Ordinary pipes have a read end and a write end. A parent process can, for example, send data to the pipe using its write end, and the child process can read it from its read end. Named pipes in UNIX are termed FIFOs.
- Windows systems also provide two forms of pipes—anonymous and named pipes. Anonymous pipes are similar to UNIX ordinary pipes. They are unidirectional and employ parent–child relationships between the communicating processes. Named pipes offer a richer form of interprocess communication than the UNIX counterpart, FIFOs.
- Two common forms of client–server communication are sockets and remote procedure calls (RPCs). Sockets allow two processes on different machines to communicate over a network. RPCs abstract the concept of function (procedure) calls in such a way that a function can be invoked on another process that may reside on a separate computer.
- The Android operating system uses RPCs as a form of interprocess communication using its binder framework.