4.3 Operations on Processes

The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism (or facility) for process creation and termination.

4.3.1 Process Creation

A process may create several new processes, via a `create-process` system call, during the course of execution. The creating process is called a parent process, whereas the new processes are called the children of that process. Each of these new processes may in turn create other processes.

In general, a process will need certain resources (such as CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or 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 subprocesses.

When a process is created it obtains, in addition to the various physical and logical resources, initialization data (or input) that may be passed along from the parent process to the child process. For example, consider a process whose function is to display the status of a file, say F1, on the screen of a terminal.

When it is created, it will get, as an input from its parent process, the name of the file F1, and it will execute using that datum to obtain the desired information.
It may also get the name of the output device. Some operating systems pass resources to child processes. On such a system, the new process may get two open files, F1 and the terminal device, and may just need to transfer the datum between the two.

When a process creates a new process, two possibilities exist in terms of execution:
1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

4.3.2 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 data (output) to its parent process (via the `wait` system call).

All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

Termination occurs under additional circumstances. A process can cause the termination of another process via an appropriate system call (for example, `abort`). Usually, only the parent of the process that is to be terminated can invoke such a system call. Otherwise, users could arbitrarily kill each other's jobs. A parent therefore needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent. A parent may terminate the execution of one of its children for a variety of reasons, such as these:

The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
The task assigned to the child is no longer required. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated.

4.4 Cooperating Process

The concurrent processes executing 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. Clearly, any process that does not share any data (temporary or persistent) with any other process is independent. On the other hand, 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.

We may want to provide an environment that allows process cooperation for several reasons:

- **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 these types of resources.

- **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. Such a speedup can be achieved only if the computer has multiple processing elements (such as CPUS or I/O channels).

- **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 have many tasks on which
to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

Concurrent execution of cooperating processes requires mechanisms that allow processes to communicate with one another and to synchronize their actions. To illustrate the concept of cooperating processes, let us 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 print program produces characters that are consumed by the printer driver. A compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.

To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. In this situation, the consumer must wait until an item is produced.

The unbounded-buffer producer-consumer problem 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 producer consumer problem 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.

The buffer may either be provided by the operating system through the use of an interprocess-communication (IPC) facility, or by explicitly coded by the application programmer with the use of shared memory.
4.5 Interprocess Communication
we showed how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a common buffer pool, and that the code for implementing the buffer be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via an interprocess communication (PC) facility.

IPC provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. IPC is particularly useful in a distributed environment where the communicating processes may reside on different computers connected with a network. An example is a chat program used on the World Wide Web.

IPC is best provided by a message-passing system, and message systems can be defined in many ways.

4.5.1 Message-Passing System

The function of a message system is to allow processes to communicate with one another without the need to resort to shared data. In this scheme, services are provided as ordinary user processes. That is, the services operate outside of the kernel. Communication among the user processes is accomplished through the passing of messages. An IPC facility provides at least the two operations: send(message) and receive(message).

Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. On the
other hand, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler.
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
- Symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or send by reference
- Fixed-sized or variable-sized messages

We look at each of these types of message systems next.

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

4.5.2.1 Direct Communication
With 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:

- \textbf{Send}(P, \text{message}) - Send a message to process P.
- \textbf{Receive} (Q, \text{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.

• Exactly one link exists between each pair of processes.

This scheme exhibits symmetry in addressing; that is, both the sender and the receiver processes must name the other to communicate. A variant of this scheme employs asymmetry in addressing. 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.

The disadvantage in both symmetric and asymmetric schemes is the limited modularity of the resulting process definitions. Changing the name of a process may necessitate examining all other process definitions. All references to the old name must be found, so that they can be modified to the new name. This situation is not desirable from the viewpoint of separate compilation.

### 4.5.2.2 Indirect Communication

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. In this scheme, a process can communicate with some other process via a number of different mailboxes.
Two processes can communicate only if they share a 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.
- A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

Now suppose that processes P1, P2, and P3 all share mailbox A. Process P1 sends a message to A, while P2 and P3 each execute a **receive** from A. Which process will receive the message sent by P1? The answer depends on the scheme that we choose:

- Allow a link to be associated with at most two processes.
- 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). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. 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 (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this 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.

On the other hand, a mailbox owned by the operating system is independent and is not attached to any particular process. 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.

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 receive privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

### 4.5.3 Synchronization

Communication between processes takes place by 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.

Different combinations of send and receive are possible.
4.5.4 Buffering

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

- **Zero capacity**: The queue has maximum length 0; 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 latter 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 has a finite capacity, however. If the link is full, the sender must block until space is available in the queue.

- **Unbounded capacity**: The queue has potentially infinite length; 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 automatic buffering.