Processes

Chapter 3

A process is a program in execution. A process is the unit of work in a modern time-sharing system. A system therefore consists of a collection of processes: operating- system processes executing system code and user processes executing user code.

3.1 Process Concept

A batch system executes jobs, whereas a time-shared system has user programs, or tasks.

3.1.1 The Process

Informally, as mentioned earlier, a process is a program in execution. A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables.

We emphasize that a program by itself is not a process. A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file). In contrast, a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

3.1.2 Process State

The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

- New. The process is being created
- Running. Instructions are being executed
- Waiting. The process is waiting for some event to occur
- Ready. The process is waiting to be assigned to a processor
- Terminated. The process has finished execution.

3.1.3 Process Control Block

Each Process is represented in the operating system by a process control block (PCB) - also called a task control block.

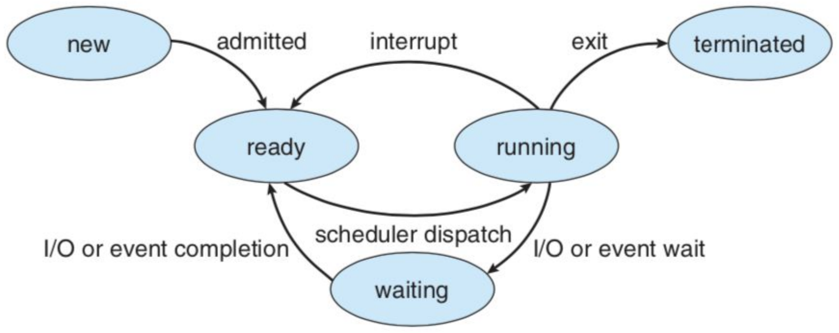

Figure 3.2 Diagram of process state.

- Process state. The state may be new, ready, running, waiting, halted, and so on.
- Program counter.The counter indicates the address of the next instruction to be executed for this process.
- CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward
- CPU-scheduling information.This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- Memory-management information. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system

3.1.4 Threads

A single thread of control allows the process to perform only one task at a time.

3.2 Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

3.2.1 Scheduling Queues

As processes enter the system, they are put into a job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue. The list of processes waiting for a particular I/O device is called a device queue.

3.2.2 Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler.

The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution. The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request.

The long-term scheduler executes much less frequently; minutes may sep- arate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of processes in mem- ory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.

The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping.

3.2.3 Context Switch

When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. Generically, we perform a state save of the current state of the CPU, be it in kernel or user mode, and then a state restore to resume operations.

A context switch is the switching of the CPU to another process requires performing a state save of the current process and a state restore of a different process. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch.

3.3 Operations on Processes

3.3.1 Process Creation

During the course of execution, a process may create several new processes. As mentioned earlier, 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. A child process 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.

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.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

3.3.2 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 a status value (typically an integer) 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.

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. (To determine whether this has occurred, the parent must 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.

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. A parent process may wait for the termination of a child process by using the wait() system call. The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated. A process that has terminated, but whose parent has not yet called wait(), is known as a zombie process. Now consider what would happen if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as orphans.

3.4 Interprocess Communication

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 several reasons for providing an environment that allows process cooperation:

- 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 such information.

- 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. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- 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 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) mech- anism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: shared memory and mes- sage passing. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange informa- tion 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.

3.4.1 Shared-Memory Systems

A producer process produces information that is consumed by a consumer process. One solution to the producer–consumer problem uses shared memory. 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. This buffer will reside in a region of memory that is shared by the producer and consumer processes. 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.
Two types of buffers can be used. The unbounded buffer 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 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.

3.4.2 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.

3.4.2.1 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.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. 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. In general, any such hard-coding techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described next.

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.

3.4.2.2 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. (Throughout this text, you will encounter the concepts of synchronous and asynchronous behavior in relation to various operating-system algorithms.)

• 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

3.4.2.3 Buffering

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.

3.6 Communication in Client-Server Systems

3.6.1 Sockets

A socket is defined as an endpoint for communication. A pair of processes communicating over a network employs a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection.

All connections must be unique. Therefore, if another process also on host X wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets.

The IP address 127.0.0.1 is a special IP address known as the loopback. When a computer refers to IP address 127.0.0.1, it is referring to itself. This mechanism allows a client and server on the same host to communicate using the TCP/IP protocol.

3.6.2 Remote Procedure Calls

A port is simply a number included at the start of a message packet. Whereas a system normally has one network address, it can have many ports within that address to differentiate the many network services it supports. If a remote process needs a service, it addresses a message to the proper port. For instance, if a system wished to allow other systems to be able to list its current users, it would have a daemon supporting such an RPC attached to a port—say, port 3027.

3.6.3 Pipes

A pipe acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations. In implementing a pipe, four issues must be considered:

1. Does the pipe allow bidirectional communication, or is communication unidirectional?
2. If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?
3. Must a relationship (such as parent – child) exist between the communi- cating processes?
4. Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

3.6.3.1 Ordinary Pipes

Ordinary pipes allow two processes to communicate in standard producer– consumer fashion: the producer writes to one end of the pipe (the write-end) and the consumer reads from the other end (the read-end). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.

An ordinary pipe cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it creates via fork().

3.6.3.2 Named Pipes

Named pipes provide a much more powerful communication tool. Com- munication can be bidirectional, and no parent–child relationship is required. Once a named pipe is established, several processes can use it for communi- cation.