PROCESS MANAGEMENT

Process Management

The operating system manages many kinds of activities ranging from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is encapsulated in a process. A process includes the complete execution context (code, data, PC, registers, OS resources in use etc.).

Process is a system abstraction, it illustrates that system has only one job to do. Every program running on a computer, be it background services or applications, is a process. As long as a von Neumann architecture is used to build computers, only one process per CPU can be run at a time. Older microcomputer OSes such as MS-DOS did not attempt to bypass this limit, with the exception of interrupt processing, and only one process could be run under them. Mainframe operating systems have had multitasking capabilities since the early 1960s. Modern operating systems enable concurrent execution of many processes at once via multitasking even with one CPU. Process management is an operating system's way of dealing with running multiple processes. Since most computers contain one processor with one core, multitasking is done by simply switching processes quickly. Depending on the operating system, as more processes run, either each time slice will become smaller or there will be a longer delay before each process is given a chance to run. Process management involves computing and distributing CPU time as well as other resources. Most operating systems allow a process to be assigned a priority which affects its allocation of CPU time. Interactive operating systems also employ some level of feedback in which the task with which the user is working receives higher priority. Interrupt driven processes will normally run at a very high priority. In many systems there is a background process, such as the System Idle Process in Windows, which will run when no other process is waiting for the CPU.

It is important to note that a process is not a program. A process is only ONE instant of a program in execution. There are many processes can be running the same program. The five major activities of an operating system in regard to process management are

- Creation and deletion of user and system processes.
- Suspension and resumption of processes.
- A mechanism for process synchronization.
- A mechanism for process communication.
- A mechanism for deadlock handling.

Kernel and process

This section presents the kernel and process that are two important parts of an operating system.

Kernel

The kernel provides the most basic interface between the machine itself and the rest of the operating system. The kernel is responsible for the management of the central processor. The kernel includes the dispatcher to allocate the central processor, to determine the cause of an interrupt and initiate its processing, and some provision for communication among the various system and user tasks currently active in the system.

The kernel is the core of an operating system. The main functions of the kernel are as:

- To provide a mechanism for the creation and deletion of process,
- To provide CPU scheduling, memory management, and device management for these process,

- To provide synchronization tools so that the processes can synchronize their actions,
- To provide communication tools so that processes can communicate with each other.

As an example, consider the Unix operating system that has two separable parts: the kernel and the service programs. The kernel provider on the file system, CPU scheduling, memory management, and other operating system functions through system calls. System calls define the programmer interface to operating system, the set of systems programs commonly available defines the user interface. These interfaces define the context the kernel must support.

System calls provide the interface between a running program and the operating system. These calls are generally available as assembly language instructions, and are usually listed in the manuals used by assembly language programmers. Some systems may allow system calls to be made directly from a higher-level language program, in which case the calls normally resemble predefined function or subroutine calls. They may generate a call to a special run-time routine that makes the system call, or the system call may be generated directly in-line. The C language allows system calls to be made directly. Some Pascal systems also provide an ability to make system calls directly from a Pascal program to the operating system.

Scheduling is a fundamental operating system function, since almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Consequently, its scheduling is often performed in the operating system.

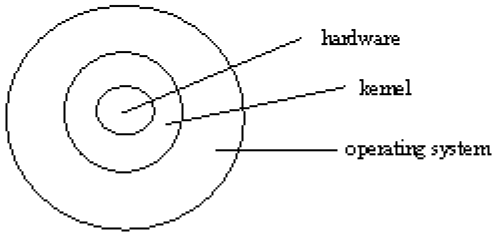


Figure 1.2 Structure of kernel-based operating system

The kernel-based design often is used for designing of the operating system. The kernel (more appropriately called the nucleus) is a collection of primitive facilities over which the rest of the operating system is built, using the functions provided by the kernel (see Fig. 1.2). Thus, a kernel provides an environment to build an operating system in which the designer has considerable flexibility because policy and optimization decisions are not made at the kernel level. An operating system is an orderly growth of software over the kernel, where all decisions regarding process scheduling, resource allocation, execution environment, file system, and resource protection etc. are made.

Consequently, a kernel is a fundamental set of primitives that allows the dynamic creation and control of process, as well as communication among them. Thus, the kernel only support the notion of processes and does not include the concept of a resource. However, as operating systems have matured in functionality and complexity, more functionality has been relegated to the kernel. A kernel should contain a minimal set of functionality that is adequate to build an operating system with a given set of objectives.

Process

The most central concept in any operating system is the concept of process: an abstraction of a running program. Everything else hinges on this concept, and it is important that the operating system designer know what a process is as early as possible.

All modern computers can do several things at the same time. While running a user program, a computer can also be reading from a disk and printing on a terminal or a printer. In a multiprogramming system, the CPU also switches from program to program, running each for tens or hundreds of milliseconds. While, strictly speaking, at any instant of time, the CPU is running only one program, in the course of one second, it may work on several programs, thus giving the users the illusion of parallelism. Sometimes people speak of pseudo-parallelism to mean this rapid switching back and forth of the CPU between program, to contrast it with the true hardware parallelism of the CPU computing while one or more I/O devices are running. Keeping track of multiple, parallel activities is not easy. Therefore, over the years operating system designers developed a model that makes parallelism easier to deal with. This model is the subject of the following paragraphs.

The Process Model

In this model, all the runnable this program, often including the operating system, is organized into a number of sequential processes, or just processes for short. A process is an executing program, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process, but to understand the system, it is much easier to think about a collection of process running in (pseudo) parallel, than to try to keep track of how the CPU switches form program to program. This rapid switching back and forth called multiprogramming, as we saw in the previous section. In Fig. 1.3(a) an example of multiprogramming with four programs in given.

In Fig. 1.3(b) we see how this is abstracted into four processes, each with its own flow of control (i.e., its own program counter), and each one running independently of the other ones. In Fig. 1.3(c) we see that over a long enough time interval, all the processes have made progress, but at any given instant only one process is actually running.

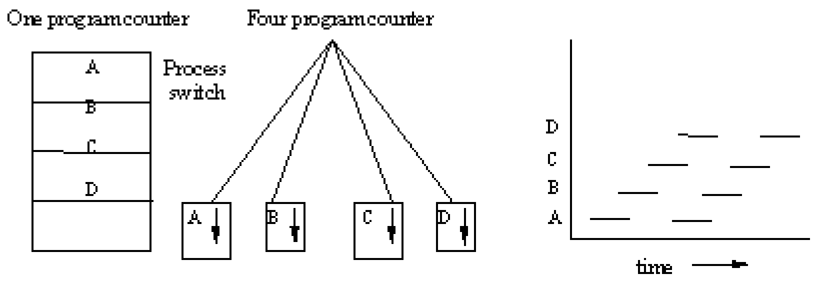


Figure1.3. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. Only one program is active at any instant.

With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform, and probably not even reproducible if the same processes are run again. Thus, processes must not be programmed with built-in assumptions about timing. Consider, for example, an

I/O process that starts to move a magnetic tape, executes an idle loop 1000 times to let the tape get up to speed, and then issues a command to read the first record. If the CPU decides to switch to another process during the idle loop, the tape process might not run correctly. When a process has critical real-time requirements like this, that is, certain events absolutely must occur within a specified number of milliseconds, special measures must be taken to ensure that they do occur. Normally, however, most processes are not affected by the underlying multiprogramming of the CPU or the relative speeds of different processes.

The difference between a process and a program is subtle, but crucial. An analogy may help make this point clearer. Consider a culinary-minded computer scientist who is baking a birthday cake for his daughter. He has a birthday cake recipe and a kitchen well-stocked with the necessary input: flour, eggs, sugar, and so on. In this analogy, the recipe is the program (i.e., an algorithm expressed in some suitable notation), the computer scientist is the processor (CPU), and the cake ingredients are the input data. The process is the activity consisting of our baker reading the recipe, fetching the ingredients, and baking the cake.

Now imagine that the computer scientist’s son comes running in crying, saying that he has been stung by a bee. The computer scientist records where he was in the recipe (the state of the current process is saved), gets out a first aid book, and begins following the directions in it. Here we see the processor being switched from one process (baking) to a higher priority process (administering medical care), each having a different program (recipe vs. first aid book). When the bee sting has been taken care of, the computer scientist goes back to his cake, continuing at the point where he left off.

The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state. A single processor may be shared among several processes, with some scheduling algorithm being used to determine when to stop work on one process and service a different one.

Process Hierarchies

Operating systems that support the process concept must provide some way to create all the processes needed. In very simple systems, or in systems designed for running only a single application, it may be possible to have all the processes that will ever be needed be present when the system comes up. In most systems, however, some way is needed to create and destroy processes as needed during operation. In operating systems, system calls exist to create a process, load into memory, and start it running. Whatever the exact nature of the system call, processes need a way to create other processes. Note that each process has one parent but zero, one, two, or more children.

Although each process is an independent entity, with its own program counter and internal state, processes often need to interact with other processes. One process may generate some output that another process uses as input.

In Fig. 4 we see a state diagram showing the three states a process may be in:

1. Running (actually using the CPU at the instant).
2. Blocked (unable to run until some external event happens).
3. Ready (runnable; temporarily stopped to let another process run).

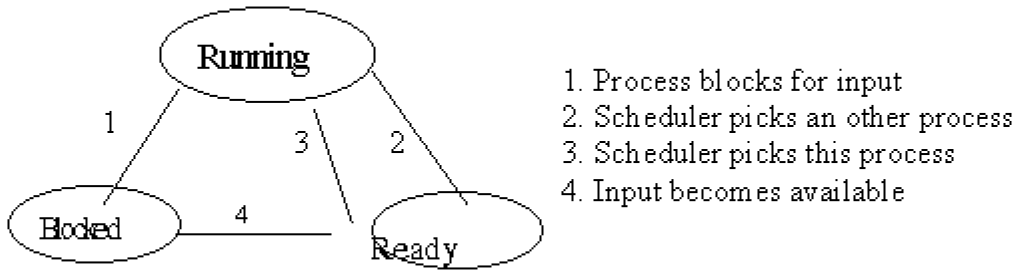

Running
Blocked
Ready
1
2
3
4
1. Process blocks for input
2. Scheduler picks an other process
3. Scheduler picks this process
4. Input becomes available

Fig. 1.4 A process can be in running, blocked or ready (also called runnable) state

Four transitions are possible among these states, as shown. Transition 1 occurs when a process discovers that it cannot continue. In some systems the process must execute a system call (see section 5), BLOCK, to get into blocked state. In other systems, when a process reads from a pipe or special file (e.g., a terminal) and there is no input available, the process is automatically blocked.

Transitions 2 and 3 are caused by the process scheduler, a pair the operating system, without the process even knowing about them. Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. Transition 3 occurs when all the other processes have had their share and it is time for the first process to run again. The subject of scheduling, that is, deciding which process should run when and for how long, is an important one; we will look at it later in this chapter. Many algorithms have been devised to try to balance the competing demands of efficiency for the systems as a whole and fairness to individual processes.

Transition 4 occurs when the external event for which a process was waiting (such as the arrival of some input) happens. If no other process is running at that instant, transition 3 will be triggered immediately, and the process will start running. Otherwise it may have to wait in *ready* state for a little while until the CPU is available.

Using the process model, it become much easier to think about what is going on inside the system. Some of the processes run programs that carry out commands typed in by a user. Other processes are part of the system and handle tasks such as carrying out requests for file services or managing the details of running a disk or a tape drive. When for example, a disk interrupt occurs, the system makes a decision to stop running the current process and run the disk process, which was blocked waiting for that interrupt. Thus, instead of thinking about interrupts, we can think about user processes, disk processes, terminal processes, and so on, which block when they are waiting for something to happen. When the disk has been read or the character typed, the process waiting for it is unblocked and is eligible to run again.

To implement the process model, the operating system maintains a table (an array of structures), called the process table, with one entry per process. This entry contains information about the process state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from running to ready state so that it can be restated later as if it had never been stopped.

In operating systems the process management, memory management, and file management are each handled by separate modules within the system, so the process table is partitioned, with each module maintaining the fields that is needs.

Inter Process Communication (IPC)

Interprocess communication (IPC) is a set of programming interfaces that allow a programmer to create and manage individual program processes that can run concurrently in an operating system. This allows a program to handle many user requests at the same time. Since even a single user request may result in multiple processes running in the operating system on the user's behalf, the processes need to communicate with each other. The IPC interfaces make this possible. Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods. IPC methods include:

i) Pipes;

ii) Named Pipes;

iii) Semaphores;

iv) Shared memory;

v) Message queuing

vi) Sockets

- **Pipes**:

In computer programming, especially in Unix operating systems, a pipe is a technique for passing information from one program process to another. Unlike other forms of interprocess communication (IPC), a pipe is one-way communication only. Basically, a pipe passes a parameter such as the output of one process to another process which accepts it as input. The system temporarily holds the piped information until it is read by the receiving process. This allows the flow of data in one direction only. Data from the output is usually buffered until the input process receives it which must have a common origin

Using a UNIX shell (the UNIX interactive command interface), a pipe is specified in a command line as a simple vertical bar (|) between two command sequences. The output or result of the first command sequence is used as the input to the second command sequence. The *pipe* system call is used in a similar way within a program.

For two-way communication between processes, two pipes can be set up, one for each direction. A limitation of pipes for interprocess communication is that the processes using pipes must have a common parent process (that is, share a common open or initiation process and exist as the result of a *fork* system call from a parent process).

A pipe is fixed in size and is usually at least 4,096 bytes.

- **Named Pipes**:

In computer programming, a named pipe is a method for passing information from one computer process to other processes using a pipe or message holding place that is given a specific name. Unlike a regular pipe, a named pipe can be used by processes that do not have to share a common process origin and the message sent to the named pipe can be read by any authorized process that knows the name of the named pipe.

A named pipe is sometimes called a "FIFO" (first in, first out) because the first data written to the pipe is the first data that is read from it.

- **Message queuing:**

In programming, message queueing is a method by which process (or program instances) can exchange or pass data using an interface to a system-managed queue of messages. Messages can vary in length and be assigned different types or usages. A message queue can be created by one process and used by multiple processes that read and/or write messages to the queue. For example, a server process can read and write messages from and to a message queue created for client processes. The message type can be used to associate a message with a particular client process even though all messages are on the same queue.

The message queue is managed by the operating system (or kernel). Application programs (or their processes) create message queues and send and receive messages using an application program interface (API). In Unix systems, the C programming language *msgget* function is used with various parameters specifying the action requested, message queue ID, message type, and so forth.

The maximum size of a message in a queue is limited by the operating system and is typically 8,192 bytes.

- **Semaphores:**

Semaphores are a technique for coordinating or synchronizing activities in which multiple processes compete for the same operating system resources. A semaphore is a value in a designated place in operating system (or kernel) storage that each process can check and then change. **Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphones can be binary (0 or 1) or can have additional values.** Typically, a process using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.

Semaphores are commonly use for two purposes: to share a common memory space and to share access to files. Semaphores are one of the techniques for interprocess communication (IPC). The C programming language provides a set of interfaces or "functions" for managing semaphores.

- **Shared Memory**:

Shared memory is a method by which program processes can exchange data more quickly than by reading and writing using the regular operating system services. For example, a client process may have data to pass to a server process that the server process is to modify and return to the client. Ordinarily, this would require the client writing to an output file (using the buffers of the operating system) and the server then reading that file as input from the buffers to its own work space. Using a designated area of shared memory, the data can be made directly accessible to both processes without having to use the system services. To put the data in shared memory, the client gets access to shared memory after checking a semaphore value, writes the data, and then resets the semaphore to signal to the server (which periodically checks shared memory for possible input) that data is waiting. In turn, the server process writes data back to the shared memory area, using the semaphore to indicate that data is ready to be read.

- **Sockets**:

Sockets is a method for communication between a client program and a server program in a network. A socket is defined as "the endpoint in a connection." Sockets are created and used with a set of programming requests or "function calls" sometimes called the sockets application programming interface (API). The most common sockets API is the Berkeley Unix C interface for sockets. Sockets can also be used for communication between processes within the same computer.

This is the typical sequence of sockets requests from a server application in the "connectionless" context of the Internet in which a server handles many client requests and does not maintain a connection longer than the serving of the immediate request:

```
socket()
|
bind()
|
recvfrom()
|
(wait for a sendto request from some client)
|
(process the sendto request)
|
sendto (in reply to the request from the client...for example, send an HTML file)
```

A corresponding client sequence of sockets requests would be:

```
socket()
|
bind()
|
sendto()
|
recvfrom()
```

Sockets can also be used for "connection-oriented" transactions with a somewhat different sequence of C language system calls or functions.

Mutual exclusion

Mutual exclusion processes has a shortcoming which is the fact that it wastes the processor time. There are primitive interprocesses that block instead of wasting the processor time.

Some of these are:

Sleep and Wakeup

SLEEP is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The WAKEUP call has one parameter, the process to be awakened.

The Producer-Consumer Problem

In this case, two processes share a common, fixed-size buffer. One of the processes puts information into the buffer, and the other one, the consumer, takes it out. This could be easy with 3 or more processes in which one wakeup waiting bit is insufficient, another patch could be made, and a second wakeup waiting bit is added of 8 or 32 but the problem of race condition will still be there.

Events Counter

This involves programming a program without requiring mutual exclusion. Event counters value can only increase and never decrease. There are three operations defined on an event counter for example, E:

1. Read (E): Return value of E
2. 2. Advance (E): Atomically increment E by 1.
3. Await (E, v): Wait until E has a value of v or more.

Two events counters are used. The first one, in would be to count the cumulative number of items that the producer discussed above has put into the buffer since the program started running. The other one out, counts the cumulative number of items that the consumer has removed from the buffer so far. Therefore, it is clear that in must be greater than or equal to out, but not more that the size of the buffer. This is method that works with pipes discussed above.

Monitors

This is about the best way of achieving mutual exclusion.

A Monitor is a collection of procedures, variables, and data structures that are grouped together in a special kind of module or package. The monitor uses the wait and signal. The "WAIT" is to indicate to the other process that the buffer is full and so causes the calling process to block and allows the process that was earlier prohibited to enter the process at this point. "SIGNAL" will allow the other process to be awakened by the process that entered during the "WAIT".

For more information on these procedures, visit

PROCESS SCHEDULING

Scheduling involves determining which thread should be run on the processor at a given time.

This is called rime slice, and its actual value depends on the system configuration.

Each thread currently has a base priority which is set by the programmer who created the program. It defines how the thread is executed in relation to other system threads, and the thread with the highest priority gets use of the processor

The scheduler is concerned with deciding on policy, not providing a mechanism. A good scheduling algorithm according to A.S. Tanenbaum and A.S. wood hull should include the following:

- Fairness- makes sure each process gets its fair share of the CPU.

- Efficiency –keep the CPU busy 100 percent of the time.
- Response time- minimise response time for interactive users.
- Turnaround – minimise the time batch users must wait for output.
- Throughput – maximise the number of jobs processed per hour.

Scheduling objective should also:

- Be predictable
- Enforce priorities
- Balance resource use
- Avoid indefinite postponement
- Degrade gracefully under heavy load

Scheduler is made up of two main parts:

Primary scheduler:

This determines the priority numbers of threads which are currently running. It then compares their priority and assigns resources to them, depending on their priority. Threads with the highest priority will be executed for the current time slice. With two or more threads with the same priority, they are put on a stack to allow each run on a given time slice.

Secondary Scheduler:

While primary scheduler runs threads with the highest priority, the secondary scheduler is responsible for increasing the priority of non-executing threads. It is important that those low-priority threads are given a chance to run on the operating system and this is the function of this type of scheduler. This will prevent the blocking of I/O operations.

A complication that schedulers have to deal with is that every process is unique and unpredictable. Some spend a lot of time waiting for file I/O, while others would use the CPU for hours at a time if given the chance. When the scheduler starts running some process, it never knows for sure how long it will be until that process blocks, either for I/O, or on a semaphore, or for some other reason. To make sure that no process runs too long, nearly all computers have an electronic timer or clock built in, which causes an interrupt periodically.

The strategy of allowing processes that are logically runnable to be temporarily suspended is called preemptive scheduling, while the run to completion method of the early batch systems is called nonpreemptive scheduling.

The scheduler operates on a queue of processes, each of which can either be:

- Running. This is where is actually currently running on the processor.
- Waiting. This is where the process is waiting on another process to run and provide and provide it with some data, or if a process is waiting to access a resource. A waiting process can sometimes turn into a zombie process, where a process terminates for some reason, but whose parent process has not yet waited for it to terminate. A zombie process is not a big problem, as it has no resources allocated to it.
- Ready. This is where the process is ready to run on the processor, and is not waiting for any other process or has terminated.
- Terminated. This is where a process has finished its run, and all resources that have been allocated to it must be taken away from it.

The scheduler thus makes a decision whenever a change occurs, such as:

- Running to waiting running to ready.
- Waiting to ready running to terminate.

A pre-emptive scheduler uses a timer to allow each process some time on the processor coordinates access to shared data. Along with this, it requires a kernel designed to protect the integrity of its own data structures.

<u>Scheduling queues</u>

There are three main system queues: Job Queue-incoming jobs, Ready Queues and Device Queues (blocked processes). Normally the type of scheduler chosen depends on the type of system that is required, such as:

- Long-term (job) scheduler. This type of scheduler is used in batch systems.
- Short-term scheduler. This type of scheduler typically uses a FIFO (First In, First Out) queue, or a priority queue.
- Medium-term scheduler. This type of scheduler swaps processes out to improve job mix.

Scheduling Algorithm:

<u>First-Come, First-Served (FCFS)</u>

This type of scheduling is used with non-pre-emptive operating systems, where the time that a process waits in the queue is totally dependent on the processes which are in front of it, as illustrated in Figure 5.3. The response of the system is thus not dependable.

<u>Shortest-Job-First (SJF)</u>

This is one of the most efficient algorithms, and involves estimating the amount of time that a process will run for, and taking the process which take the shortest time to complete.

Priority Scheduling

This is the typical used in general-purpose operating systems (Microsoft Windows and UNIX). It can be used with either pre-emptive operating system. The main problem is to assign a priority to each process, where priorities can either be internal or external. Internal priorities related to measurable system resources, such as time limits, memory requirements, file input/output, and so on. A problem is that some processes might never get the required priority and may never get time on the processor. To overcome this, low-priority waiting processes have their priority increased, over time (Known as ageing).

Round-Robin (RR)

This is first-come, first served with pre-emption, where each process is given a finite time slice on the processor. The queue is circular, thus when a process has been run, the queue pointer is moved to the next process, as illustrated in Figure 5.4. This is a relatively smooth schedule and gives processes a share of the processor.

Multilevel Queue Scheduling

This scheme supports several different queues, and sets priorities for them. For example, a system could run two different queues: foreground (interactive) and background (batch), as illustrated in Figure5.5. The foreground task could be given a much higher priority over the background task, such as 80%-20%. Each of the queues can be assigned different priorities. Windows NT runs a pre-emptive scheme where certain system processes are given a higher priority than other non-system processes. An example priority might be (in order of priority):

System processes -Top priority

This must have top priority as the system could act unreliably if they were not executed with a given time.

Interactive processes

These are processes that require some user input, such as from the keyboard or mouse. It is important that user must feel that these processes are running with a high priority, otherwise they may try to delete them, and try to rerun the process.

Interactive editing processes

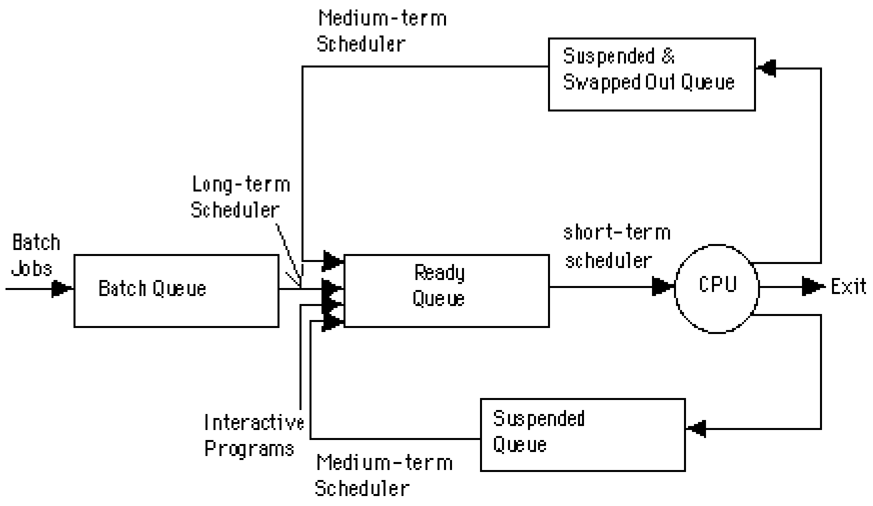
These processes tend to run without user input for long periods, but occasionally require some guidance on how they run.

Batch processes-Lowest priority

These tend to be less important processes that do not require any user input.

Multilevel Feedback Queue Scheduling

This scheme is the most complex, and allows processes to move between a number of different priority queues. Each queue has an associated scheduling algorithm. To support this there must be a way to promote and relegate processes for their current queues.



Source :

http://homepages.uel.ac.uk/u0209451/scheduler.htm

http://whatis.techtarget.com/definition/0,,sid9_gci214032,00.html

ANDREW S TANENBAUM MODERN OPERATING PRENTICE HALL INTERNATIONAL EDITIONS (1992)