SHRI VISHNU ENGINEERING COLLEGE FOR WOMEN :: BHIMAVARAM
(Autonomous)

OPERATING SYSTEMS

Unit – II :: Process Management

2.1. Process Concept

Early computer systems allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs, and these needs resulted in the notion of a process, ***which is a program in execution***. ***A process is the unit of work in a modern time-sharing system.***

A user may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package. And even if the user can execute only one program at a time, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them processes.

- ✓ 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.
- ✓ A process may also include a heap, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure.

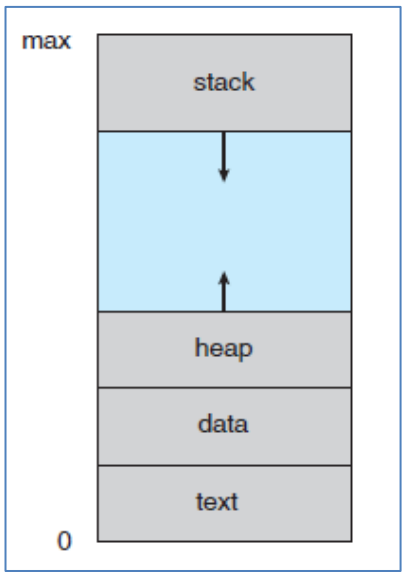


Figure : Process in Memory

2.1.1. Process States:

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. Each 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 (such as an I/O completion or reception of a signal)
- ✓ ***Ready*** - The process is waiting to be assigned to a processor
- ✓ ***Terminated*** - The process has finished execution

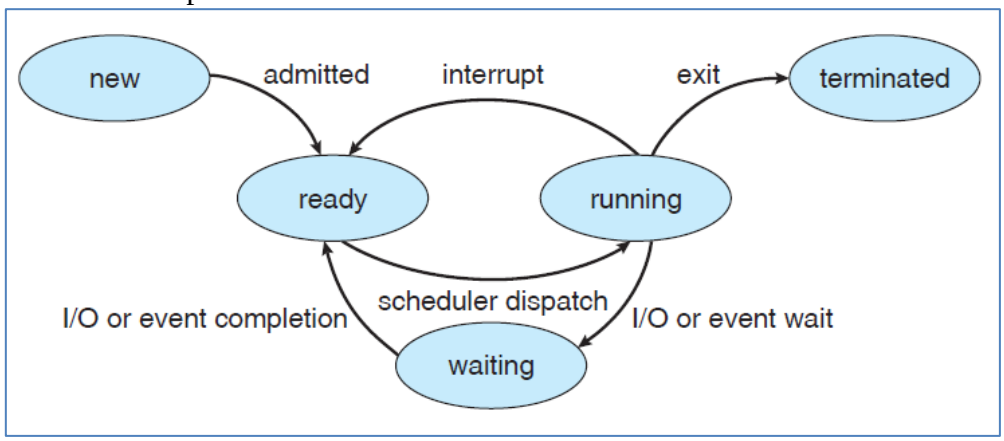


Figure : Process States

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also delineate process states more finely. ***It is important to realize that only one process can be running on any processor at any instant***. Many processes may be ready and waiting, however. The state diagram corresponding to these states is presented in above Figure.

2.1.2. Process Control Block:

Each process is represented in the operating system by a process control block (PCB)—also called ***a task control block***. A PCB is shown in Figure, contains many pieces of information associated with a specific process, including these:

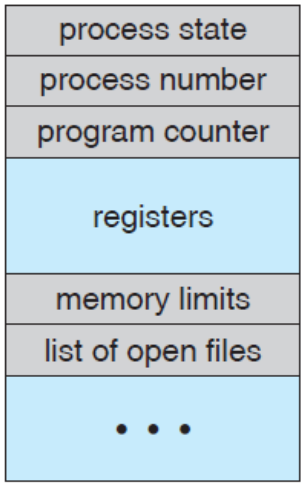


Figure : Process Control Block

- ✓ ***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 (Figure).

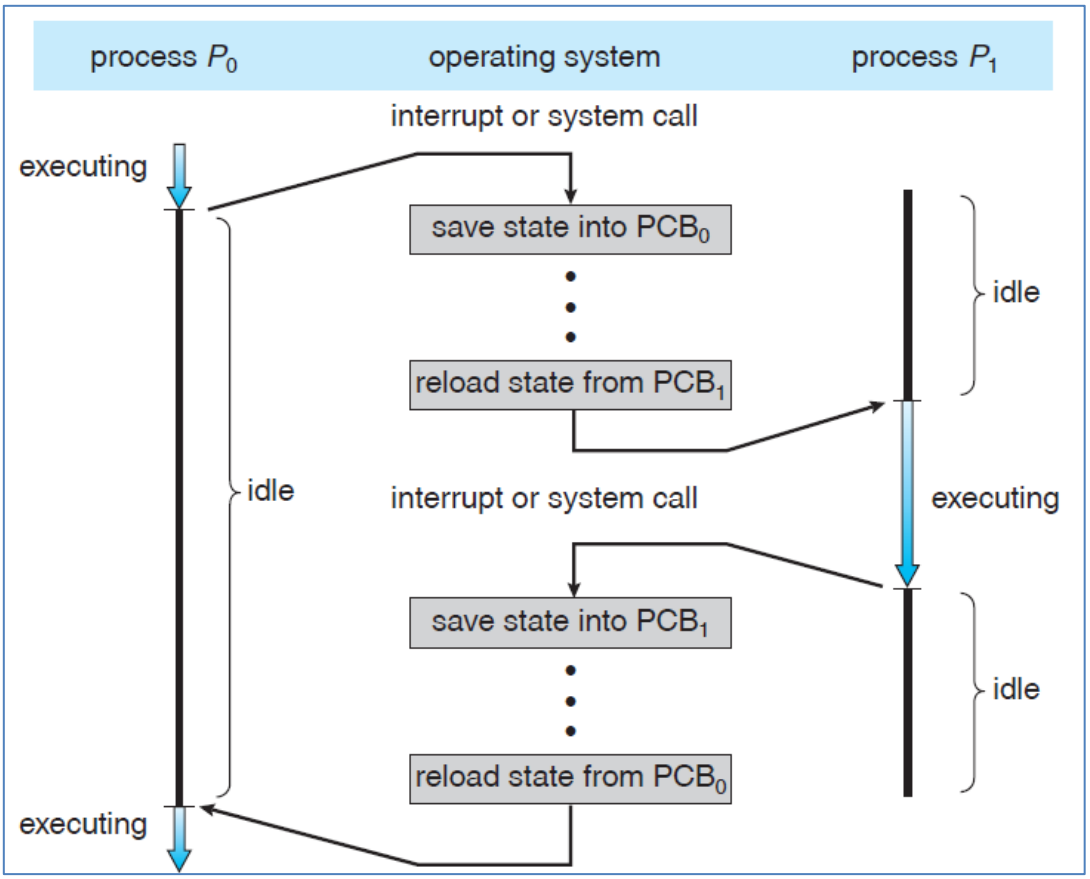


Figure : CPU switch from process to process

- ✓ ***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 information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

- ✓ ***Accounting information*** **-** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

- ✓ ***I/O status information*** - This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for any information that may vary from process to process.

2.1.3. Threads:

The process model has implied that a process is a program that performs a single thread of execution.

For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process.

Many modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

2.2. 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.

2.2.1. 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 system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a ***device queue***. Each device has its own device queue (Figure).

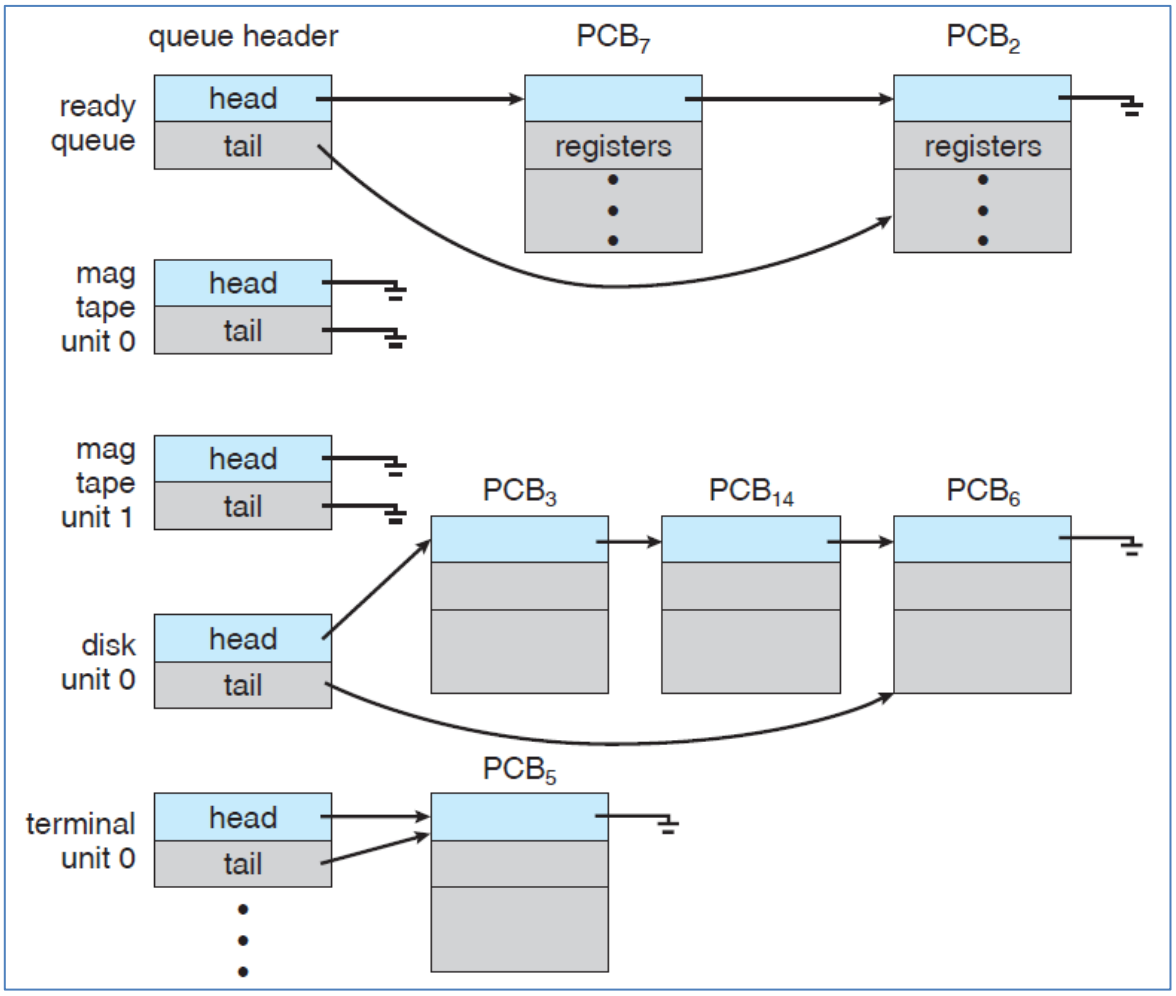


Figure : The ready queue and various I/O device queues.

A common representation of process scheduling is a queueing diagram, such as that in the following Figure. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

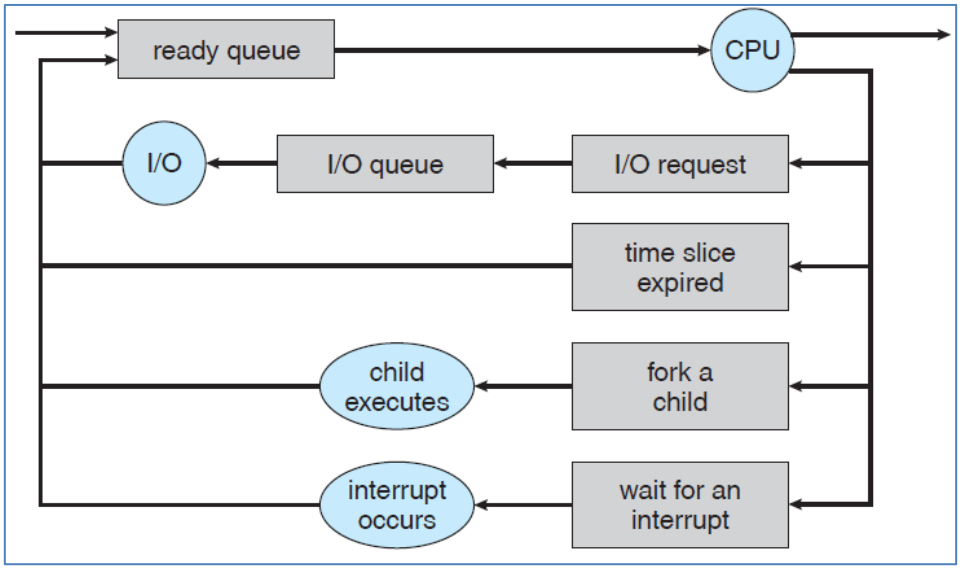


Figure : Queueing-diagram representation of process scheduling

A new process is initially put in the ready queue. It waits there until it is selected for execution, or is dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:

- ✓ The process could issue an I/O request and then be placed in an I/O queue.
- ✓ The process could create a new sub-process and wait for the sub-process's termination.
- ✓ The process could be removed forcibly from the CPU as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

2.2.2. 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.

Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. 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. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast.
- ➢ The ***long-term scheduler*** executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming. 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. Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This ***medium-term scheduler*** is diagrammed in Figure. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes 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. The process is swapped

out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

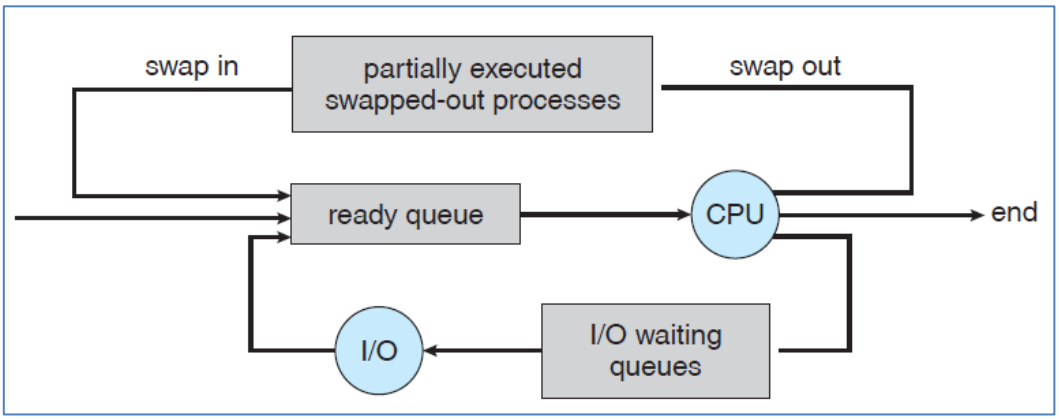


Figure : Addition of Medium-term scheduling to the queueing diagram

2.2.3. 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.

The context is represented in the PCB of the process; it includes the value of the CPU registers, the process state, and memory-management information. 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.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a ***context switch***. 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. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Context-switch times are highly dependent on hardware support.

2.3. Operations

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for ***process creation*** and ***process termination***.

2.3.1. 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, 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.

Most operating systems (including UNIX and the Windows family of operating systems) identify processes according to a unique process identifier. Most operating systems (including UNIX and the Windows family of operating systems) identify processes according to a unique process identifier. (or pid), which is typically an integer number.

In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a sub-process, that sub-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. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many sub-processes.

When a process creates a new process, two possibilities exist for execution:

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 possibilities for the address space of 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.

In UNIX, each process is identified by its process identifier, which is a unique integer. A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

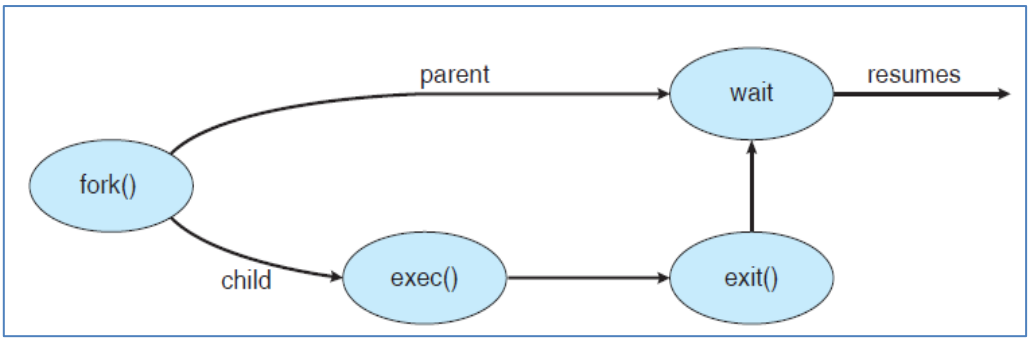


Figure : Process Creation using fork( ) system call

Typically, the exec() system call is used after a fork() system call by one of the two processes to replace the process's memory space with a new program. The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait() system call to move itself off the ready queue until the termination of the child.

2.3.2. 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 de-allocated by the operating system.

Termination can occur in other circumstances as well.

- ✓ A process can cause the termination of another process via an appropriate system call. Usually, such a system call can be invoked only by the parent of the process that is to be terminated.

- ✓ Otherwise, users could arbitrarily kill each other's jobs. Note that a parent 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.

- ✓ 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.

To illustrate process execution and termination, consider that, in UNIX, we can terminate a process by using the exit() system call; its parent process may wait for the termination of a child process by using the wait() system call.

The wait() system call returns the process identifier of a terminated child so that the parent can tell which of its children has terminated. If the parent terminates, however, all its children have assigned as their new parent the init process. Thus, the children still have a parent to collect their status and execution statistics.

2.4. Inter Process Communication

- ✓ Processes executing concurrently 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. 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 ex, 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. Such a speedup can be achieved only if the computer has multiple processing elements.

- ✓ ***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, printing, and compiling in parallel.

Cooperating processes require an Inter-Process Communication (IPC) mechanism that will allow them to exchange data and information.

There are two fundamental models of Inter-Process Communication:

(1) Shared Memory

(2) 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***.

2.4.1. Shared Memory

Inter Process communication using shared memory requires communicating processes to establish a region of shared memory as shown in Figure.

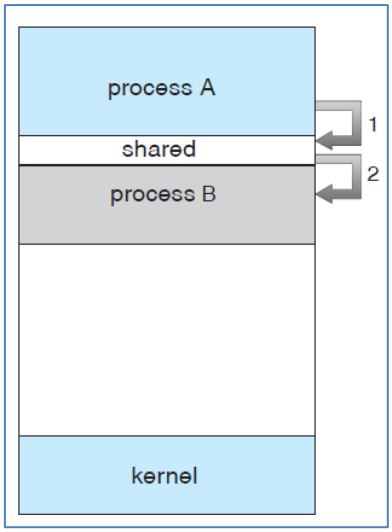


Figure : Communication Models - Shared Memory

- ✓ Typically, 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.
- ✓ Recall that, normally, the operating system 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 operating system's control.
- ✓ The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let's 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 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. The producer–consumer problem also provides a useful metaphor for the client–server paradigm.

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.

One issue this illustration does not address concerns the situation in which both the producer process and the consumer process attempt to access the shared buffer concurrently.

2.4.2. Message Passing

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a ***distributed environment***, where the communicating processes may reside on different computers connected by a network.

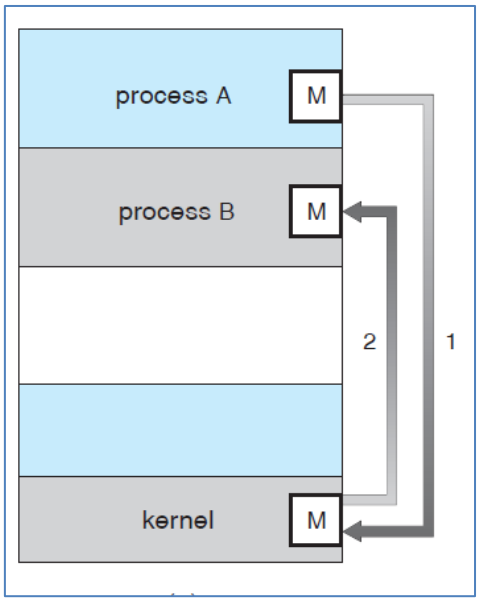


Figure : Communication Models – Message Passing

For example, a chat program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations:

- ✓ send (message)
- ✓ 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.

Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating system design.

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
- ✓ Synchronous or asynchronous communication
- ✓ Automatic or explicit buffering

There are several issues related for the implementation of link:

(a) Naming
(b) Synchronization
(c) Buffering

a. ***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. 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.

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 the processes have a shared mailbox, however. 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.
- ✓ Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

b. 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.***

- ✓ ***Blocking send -*** The sending process is blocked until the message is received by the receiving process or by the mailbox.
- ✓ ***Non-blocking send -*** The sending process sends the message and resumes operation.
- ✓ ***Blocking receive -*** The receiver blocks until a message is available.
- ✓ ***Non-blocking 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. The solution to the producer–consumer problem becomes trivial when we use blocking send() and receive() statements.

The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive(), it blocks until a message is available.

c. Buffering

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

- ✓ ***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.

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.

2.4.3. Example of IPC Systems: Windows

The Windows operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows provides support for multiple operating environments, or subsystems, with which application programs communicate via a message-passing mechanism.

The message-passing facility in Windows is called the local procedure-call (LPC) facility. The LPC in Windows communicates between two processes on the same machine. It is similar to the standard RPC mechanism that is widely used, but it is optimized for and specific to Windows.

Connection ports are named objects and are visible to all processes; they give applications away to set up communication channels. The communication works as follows:

- ✓ The client opens a handle to the subsystem's connection port object.
- ✓ The client sends a connection request.
- ✓ The server creates two private communication ports and returns the handle to one of them to the client.
- ✓ The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Windows uses ***two types of message-passing*** techniques over a port that the client specifies when it establishes the channel. The simplest, which is used for ***small messages***, uses the port's message queue as intermediate storage and copies the message from one process to the other. Under this method, messages of up to 4KB can be sent.

If a client needs to send a ***larger message***, it passes the message through a section object, which sets up a region of shared memory. The client has to decide when it sets up the channel whether or not it will need to send a large message.

In both cases, a callback mechanism can be used when either the client or the server cannot respond immediately to a request. The callback mechanism allows them to perform asynchronous message handling.

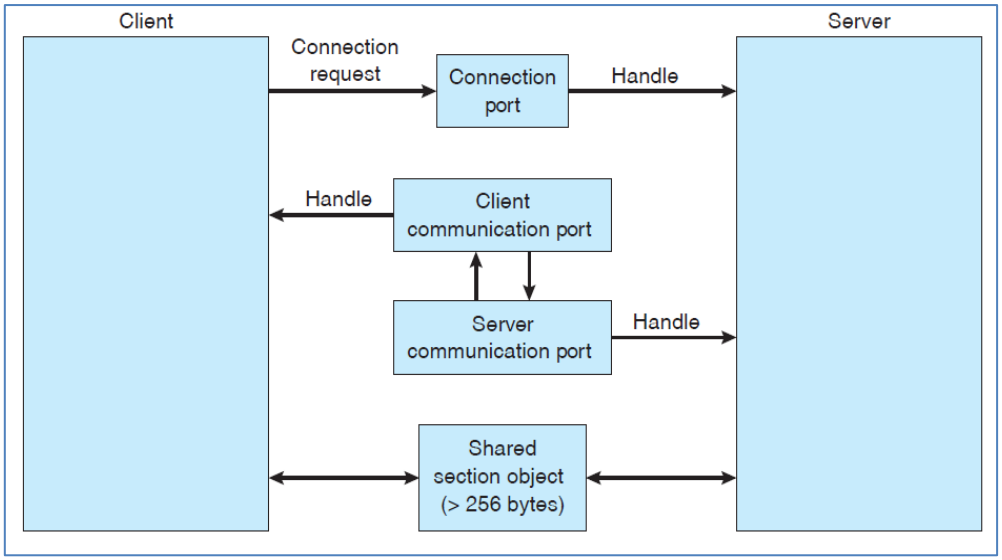


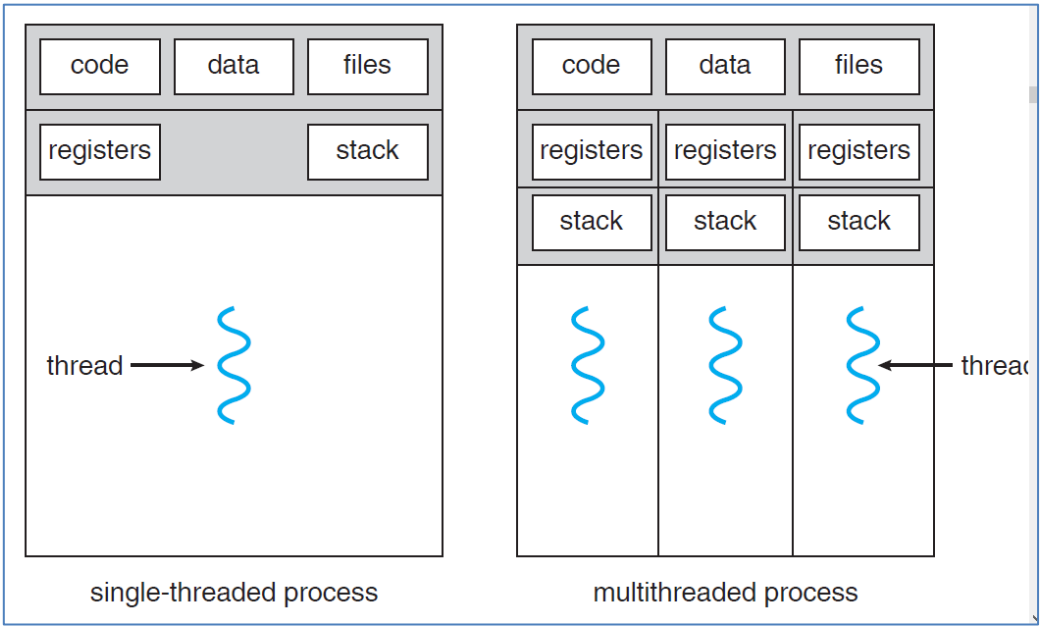
Figure : Local Procedure calls in Windows

2.5. Multi Thread Programming Model

Most modern operating systems now provide features enabling a process to contain multiple threads of control.

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

A traditional (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure illustrates the difference between a traditional single-threaded process and a multithreaded process.



The benefits of multithreaded programming can be broken down into four major categories:

- ✓ ***Responsiveness -*** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded Web browser could allow user interaction in one thread while an image was being loaded in another thread.

- ✓ ***Resource sharing -*** Processes may only share resources through techniques such as shared memory or message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process towhich they belong by default. The benefit of sharing code and data is that it allows an application to
- ✓ have several different threads of activity within the same address space.

- ✓ ***Economy -*** Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is much more time consuming to create and manage processes than threads.

- ✓ ***Scalability -*** The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single-threaded process can only run on one processor, regardless how many are available. Multithreading on a multi- CPU machine increases parallelism.

Threads may be provided either at the user level, for ***user threads***, or by the kernel, for ***kernel threads***. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Ultimately, a relationship must exist between user threads and kernel threads. The three common ways of establishing such a relationship:

(a) Many-to-One Model

The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors. Green threads—a thread library available for Solaris—uses this model.

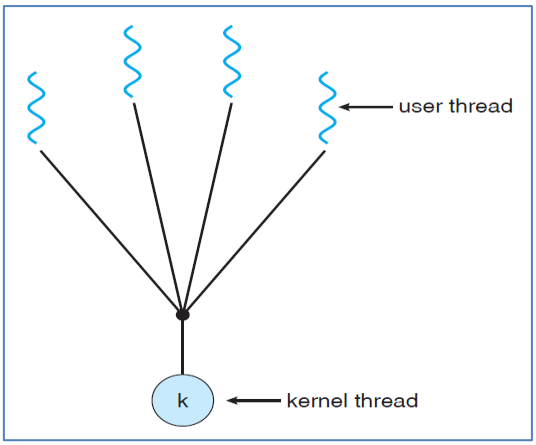


Figure : Many-to-One Model

(b) One-to-One Model

The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors.

The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.

Linux, along with the family of Windows, implement the one-to-one model.

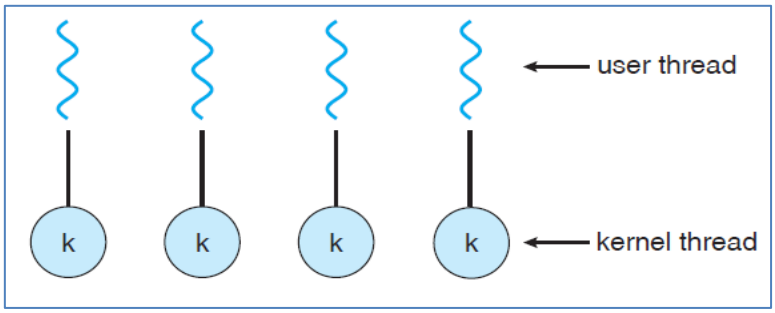


Figure : One-to-One Model

(c) Many-to-Many Model

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor).

Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.

The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create).

The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

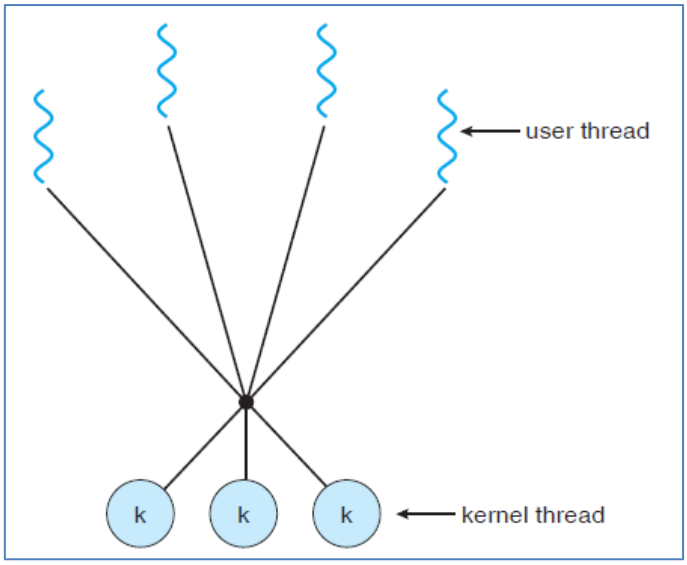


Figure : Many-to-many Model

2.6. Process Scheduling Criteria and Algorithms and their evaluation

In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. The idea is relatively simple.

A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished.

With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU. Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

(a) CPU–I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution. The durations of CPU bursts have been measured extensively.

(b) CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process. Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue.

(c) Preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes)

2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)

3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)

4. When a process terminates

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is non-preemptive or cooperative; otherwise, it is preemptive. Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

(d) Dispatcher

Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

1. Switching context
2. Switching to user mode
3. Jumping to the proper location in the user program to restart the program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

2.6.1 Process Scheduling Criteria:

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- ✓ ***CPU utilization -*** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

- ✓ ***Throughput -*** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

- ✓ ***Turnaround time -*** From the point of view of a particular process, the important criterion is howlong it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround

time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

- ✓ ***Waiting time*** - The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

- ✓ ***Response time*** - In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average.

2.6.2 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms.

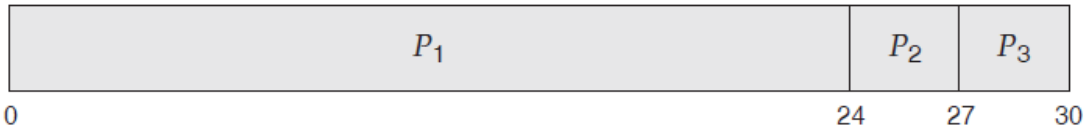
a. First-Come, First-Served Scheduling (FCFS):

By far the simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

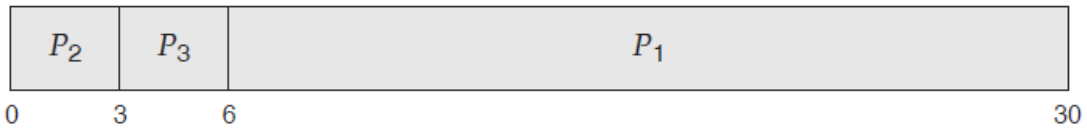
On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2, and 27 milliseconds for process P3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes CPU burst times vary greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device.

All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a convoy effect as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Note also that the FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

b. Shortest-Job-First Scheduling(SJF):

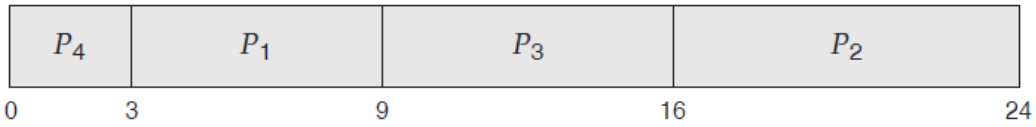
A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

Note that a more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

The real difficulty with the SJF algorithm knows the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job. Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response. (Too low a value will cause a time-limit-exceeded error and require resubmission.) SJF scheduling is used frequently in long-term scheduling.

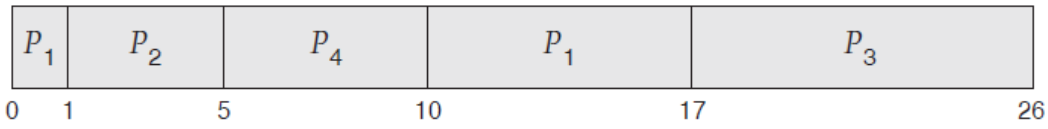
Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. With short-term scheduling, there is no way to know the length of the next CPU burst. One approach is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The SJF algorithm can be either preemptive or non-preemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled. The average waiting time for this example is [(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = 6.5 milliseconds. Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

c. Priority

The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

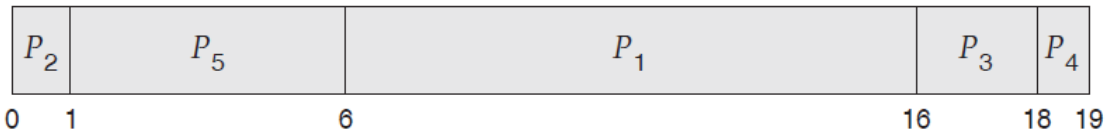
An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Note that we discuss scheduling in terms of high priority and low priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, · · ·, P5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run, or the computer system will eventually crash and lose all unfinished low-priority processes.

A solution to the problem of indefinite blockage of low-priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

d. RR

The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue.

The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process

from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

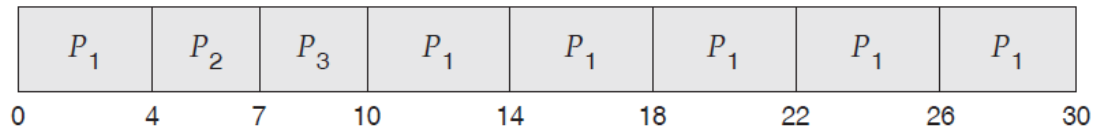
One of two things will then happen.

- The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.
- Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2. Process P2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is as follows:



Let's calculate the average waiting time for the above schedule. P1 waits for 6 milliseconds (10 - 4), P2 waits for 4 milliseconds, and P3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units. Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach is called processor sharing and (in theory) creates the appearance that each of n processes has its own processor running at 1/n the speed of the real processor. This approach was used in Control Data Corporation (CDC) hardware to implement ten peripheral processors with only one set of hardware and ten sets of registers.

The hardware executes one instruction for one set of registers, then goes on to the next. This cycle continues, resulting in ten slow processors rather than one fast one.

In software, we need also to consider the effect of context switching on the performance of RR scheduling. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure).

Thus, we want the time quantum to be large with respect to the context switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching. In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds. The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

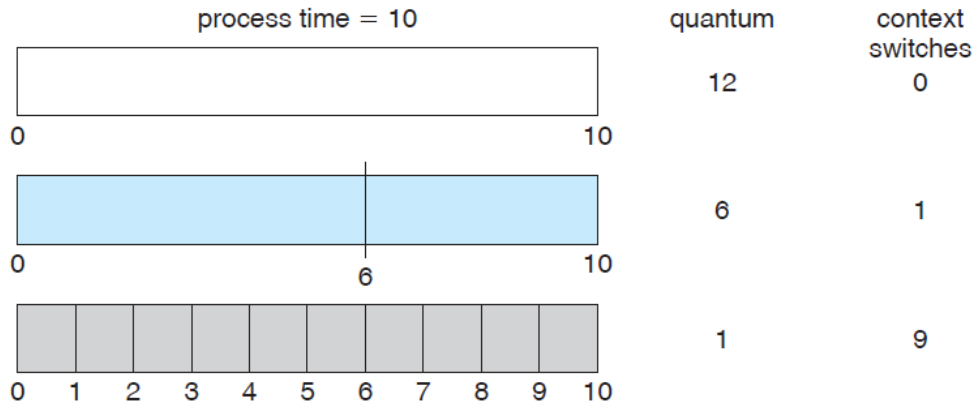


Figure : How a smaller time quantum increases context switches

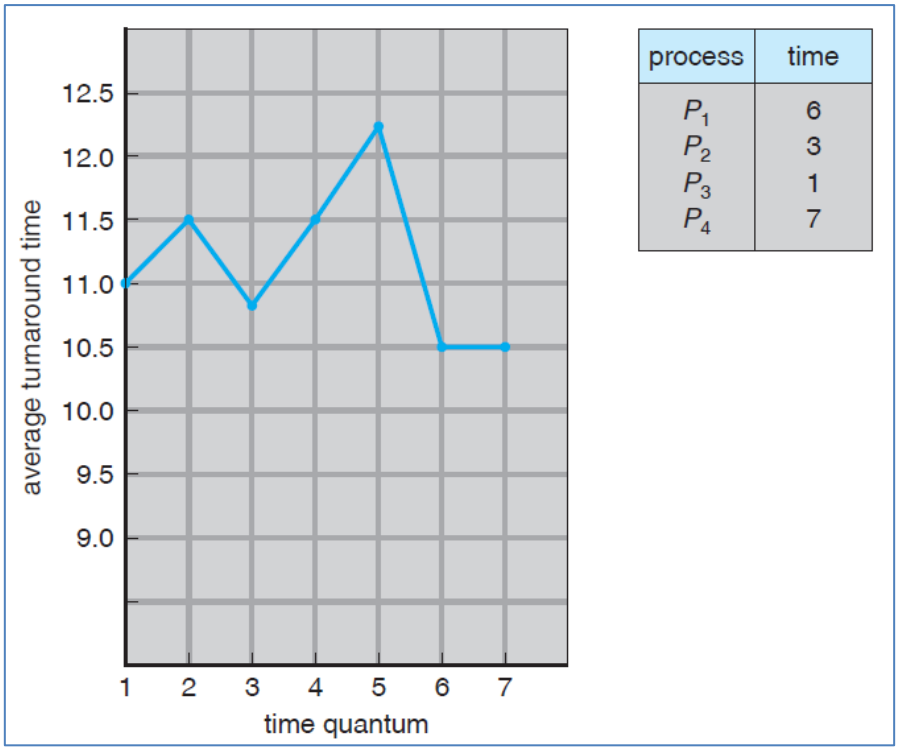


Figure : How turnaround time varies with the time quantum

Turnaround time also depends on the size of the time quantum. As we can see from Figure, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Although the time quantum should be large compared with the context switch time, it should not be too large. If the time quantum is too large, RR scheduling degenerates to an FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

e. Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common distribution is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues (Figure). The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm.

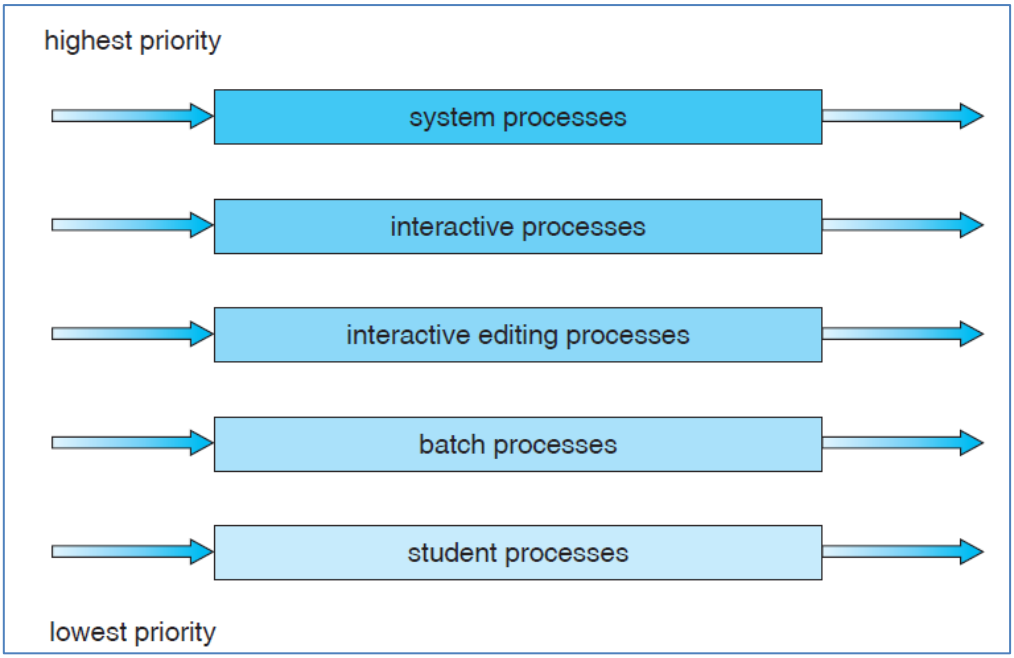


Figure : Multilevel Queue Scheduling

For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

Evaluation of Scheduling Algorithms:

CPU Scheduling involves many different scheduling algorithms which have their Advantages and Disadvantages.

1. First Come First Serve (FCFS):

Advantages:

- ✓ It is simple and easy to understand.

Disadvantages:

- ✓ The process with less execution time suffers i.e. waiting time is often quite long.
- ✓ Favors CPU Bound process then I/O bound process.
- ✓ Here, the first process will get the CPU first, other processes can get the CPU only after the current process has finished its execution. Now, suppose the first process has a large burst time, and other processes have less burst time, then the processes will have to wait more unnecessarily, this will result in more average waiting time, i.e., Convey effect.
- ✓ This effect results in lower CPU and device utilization.
- ✓ FCFS algorithm is particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

2. Shortest Job First (SJF) [Preemptive and Non- Preemptive]:

Advantages:

- ✓ Shortest jobs are favored.
- ✓ It is probably optimal, in that it gives the minimum average waiting time for a given set of processes.

Disadvantages:

- ✓ SJF may cause starvation if shorter processes keep coming. This problem is solved by aging.
- ✓ It cannot be implemented at the level of short-term CPU scheduling.

3. Round Robin (RR):

Advantages:

- ✓ Every process gets an equal share of the CPU.
- ✓ RR is cyclic in nature, so there is no starvation.

Disadvantages:

- ✓ Setting the quantum too short increases the overhead and lowers the CPU efficiency, but setting it too long may cause a poor response to short processes.
- ✓ The average waiting time under the RR policy is often long.
- ✓ If time quantum is very high then RR degrades to FCFS.

4. Priority Based (PB):

Advantages:

- ✓ This provides a good mechanism where the relative importance of each process may be precisely defined.

Disadvantages:

- ✓ If high-priority processes use up a lot of CPU time, lower-priority processes may starve and be postponed indefinitely. The situation where a process never gets scheduled to run is called starvation.
- ✓ Another problem is deciding which process gets which priority level assigned to it.

5. Multilevel Queue Scheduling (MQS):

Advantages:

- ✓ Application of separate scheduling for various kinds of processes is possible.

Disadvantages:

- ✓ The lowest level process faces the starvation problem.

*****