

Chapter 2: Process and Threads Management

Process

Process Model

Process States

Process Control Block

Context Switch

Operation on Process

- Process Creation

- Process Termination

Threads

- Benefits

- Difference between thread and process

Types of Threads

- User Level Threads

- Kernel Level Threads

Exercises

The objective of this chapter is to explain how a process is created and how processes are managed. In computer, when we click any icon (file icon or application), the application starts up. Anytime when we run a program (application), it needs to be loaded in main memory. Active Program may interact with Input Output devices (Keyboard, Mouse, Display, Printer). Active or Running program is called as a process. Process can be defined as:

[Process Definition] A process is a program in execution.

A process require resources like CPU time, memory, files, and I/O devices at the time of execution. These resources are allocated to the process either when it is created or while it is executing. Programs are executed sequentially. But in real time one processor is shared amongst many running programs (Multitasking). So one running program may be blocked and resume its work after sometime. The operating system is responsible *Process Management*. It handles creation and deletion of processes, scheduling, synchronization, communication, and deadlock handling for processes. It may happen that two or more processes are instantiated with same program. For example when we open multiple files in editor, one process is responsible for handling each file.

Process

To understand how process is created, lets imagine that we have written a program called a.c in C. On execution, this program may read in some data and output some data. When we save this program as a.c, it is simple file stored in hard disk . It has no dynamics of its own means, it cannot cause any input processing or output to happen. When we compile a C language program we get an executable file. After that when we run this program, it is loaded in main memory. The processor process instructions in program and it may interact with I/O devices.

[Difference between Program and Process] Program is a text script, a program in execution is a process. In other words, A process is an executable entity – it's a program in execution.

Operating system maintains information about process in *Process Table*. Process table contains the all information required to run , block and resume process(In multiprogramming/multitasking OS process may blocked or suspended).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.

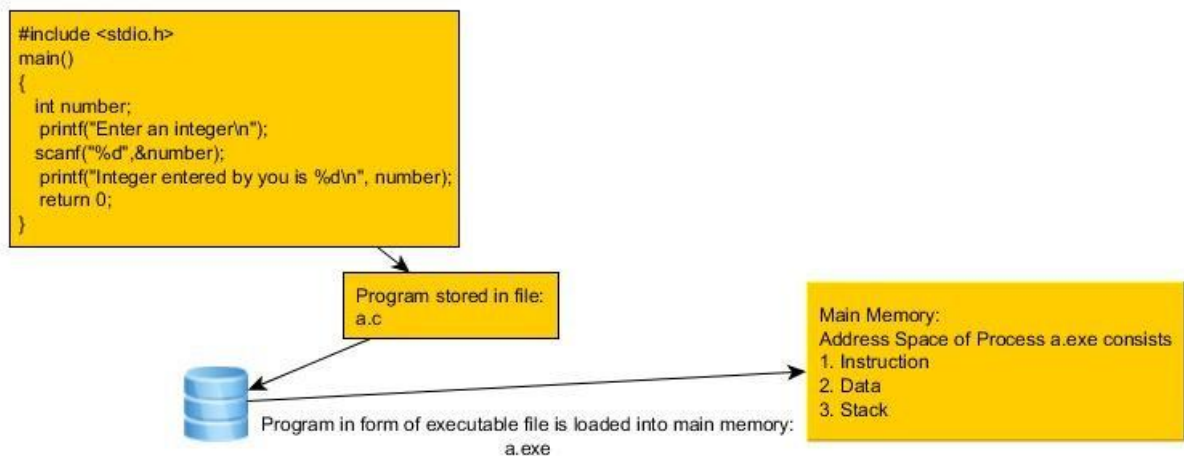


Figure 2.1: Program vs Process

Process Model

In this model, all programs running on the computer, including the operating system, are considered as processes. A process is an instance of an executing program. We can think each process execute sequentially but in reality, the real CPU switches back and forth from process to process. To understand the system, it is much easier to think about a collection of processes running in parallel than to try to keep track of how the CPU switches from program to program. This rapid switching is called multiprogramming.

Degree of Multiprogramming When multiprogramming is used, the CPU utilization can be improved. The number of process loaded in memory are called as *Degree of Multiprogramming*. If process computes only 20% of the time and other time it is spending in I/O or in other activity. We can load five processes in memory to utilize CPU. Then we will be able to utilize CPU 100 %. Because when process goes for I/O other process will be ready to use CPU. In reality all five processes may wait for I/O at the same time.

A better model can be considering probability. If a process is having p portion of I/O. If n processes is loaded in memory. We can observe that CPU utilization can be increased with increasing Degree of Multiprogramming. The CPU utilization is then given by the formula:

$$CPUWaste = p^n$$

$$CPUUtilization = 1 - CPUWaste$$

$$CPUUtilization = 1 - p^n$$

For example if processes spend 80% of their time in I/O. If we load 10 processes in memory then CPU waste will be around 10%. We will be able to get CPU Utilization up to 90%

$$CPUWaste = .8^{10}$$

$$CPUWaste = .1$$

$$CPU Utilization = 1 - .1$$

$$CPU Utilization = .9$$

This model is only approximation, In reality we will not get increase in CPU Utilization with increase in Degree of Multiprogramming after certain limit. There are few reason:

1. When one process goes for I/O and other process is allocated to CPU. The time required in this activity is not used in processing. This time to schedule other process is *Context Switch*
2. We are assuming all n processes are independent. It is possible that one process is waiting for others to complete or may waiting for resource which is in use by other process. It may not be possible sometime to completely run process independently.
3. The size of main memory also limits Degree of Multiprogramming.

Process States

When we run a process, It may be running or blocked or terminated. Process state is current operating condition. Each process may be in one of the following states:

New The process is being created. OS still not scheduled process to execute.

Ready It enters the ready state when it is considered for scheduling. The process is waiting to be assigned to a processor.

Running When a processor is available then one of the processes in "Ready" state may be chosen to run. It moves to the state "Running". Instructions of process are being executed.

Waiting The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

Terminated The process has finished execution.

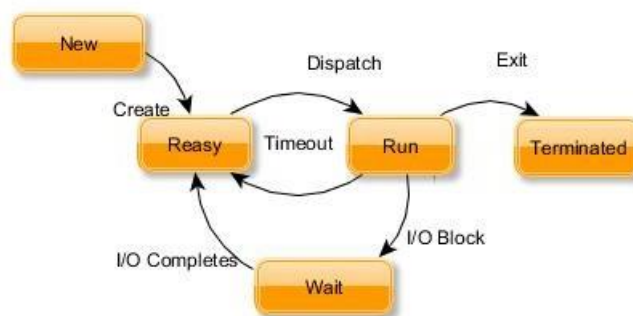


Figure 2.2: Process States

Process Control Block

Each process is represented in the operating system by a process control block (PCB). OS maintain all information about process like it's state, list of open files, current instruction being executed in a data structures for management of processes. This data structure is PCB. It is also known as task control block. It contains many pieces of information associated with a specific process, including these:

Process state : The state may be new, ready, running, waiting and terminated.

Program counter : The counter indicates the address of the next instruction to be executed for this process.

CPU register : The computer uses registers as high speed temporary storage. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.

Scheduling Information : This information includes a process priority and other scheduling parameters.

Memory-management : Info This includes information about memory allocate to process. Example is value of the base and limit registers which indicate starting and ending memory address of process.

Accounting Information : This information includes the amount of CPU and real time used, time limits.

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.

pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
⋮	

Figure 2.3: Process Control Block

Context Switch

A context switch occurs when a computer's CPU switches from one process or thread to a different process or thread. This allow for one CPU to handle multiple processes without the need for additional processors. Multitasking operating system uses context switch to allow different processes to run at the same time.

[Context Switch Definition] A context switch is the process of storing and restoring the state (context²) of a process or thread so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU and is an essential feature of a multitasking operating system. What constitutes the context is determined by the processor and the operating system.

Typically, there are three situations that a context switch is necessary:

Multitasking When the CPU needs to switch processes in and out of memory, so that more than one process can be running.

Kernel/User Switch When switching between user mode to kernel mode.

Interrupts When the CPU is interrupted to return data from a disk read.

In context switching OS stores and restores process state(containing contents of registers, program counter and stack). Process state is generally contained in *PCB(Process Control Block)*.Switching from one process to another requires a certain amount of time for doing the administration – saving and loading registers and memory maps, updating various tables and lists etc. Context switches are usually computationally intensive. The time required for context switch is overhead because it is not used in any processing. Most operating systems optimize context switches time to improve performance.

Context switching can be performed primarily by software or hardware. Some processors, like the Intel 80386 and its successor, have hardware support for context switches. They provide

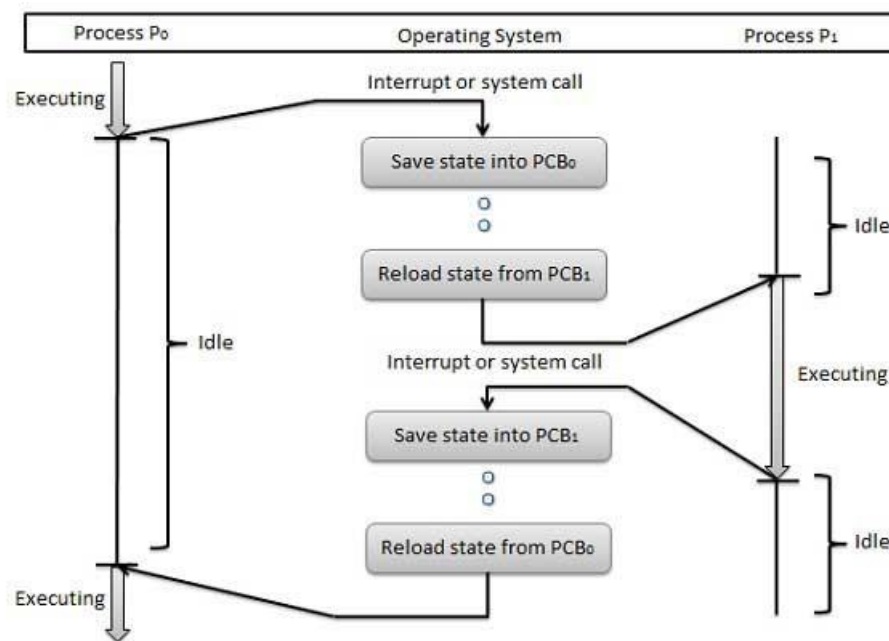


Figure 2.4: Process Context Switch

use of a special data segment designated the task state segment or TSS. When a task switch occurs the CPU can automatically load the new state from the TSS. Mainstream operating systems, including Windows and Linux, do not use this feature. Hardware context switching does not save all the registers (only general purpose registers). Software context switch is implemented in software (OS). This switching can be selective and store only those registers that need storing, whereas hardware context switching stores nearly all registers whether they are required or not.

Operation on Process

The processes can execute concurrently, and they may be created and destroyed dynamically. The operating system provides a mechanism for process creation and termination.

Process Creation

A process may create several new processes. Process can create new process with help of a create-process system call. The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes can create other processes, forming a tree of processes. There are four principal events that cause a process to be created:

1. System initialization.
2. Execution of process creation system call by a running process.
3. a user request to create a new process.
4. Initiation of a batch job.

When an operating system is booted, typically several processes are created. Some of these are foreground processes, that interacts with a (human) user and perform work for them. Other are background processes, which are not associated with particular users, but instead have some specific function. For example, one background process may be designed to accept incoming e-mails, sleeping most of the day but suddenly springing to life when an incoming e-mail arrives. Another background process may be designed to accept an incoming request for web pages hosted on the machine, waking up when a request arrives to service that request.

Process creation in UNIX and Linux are done through `fork()` or `clone()` system calls. There are several steps involved in process creation. The first step is the validation of whether the parent process has sufficient authorization to create a process. Upon successful validation, the parent process is copied almost entirely, with changes only to the unique process id, parent process, and user-space. Each new process gets its own user space.

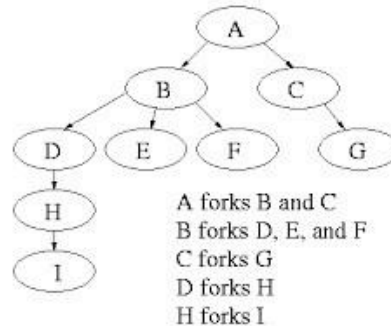


Figure 2.5: Process Hierarchy

Process Termination

A process terminates when it finishes executing its last statement. Its resources are returned to the system. It is deleted from any system lists or tables, and its process control block (PCB) is erased. Its memory space is returned to a free memory pool. The new process terminates the existing process, usually due to following reasons:

1. Normal Exist: Most processes terminates because they have done their job. This call is exist in UNIX.
2. Error Exist: When process discovers a fatal error. For example, a user tries to compile a program that does not exist.
3. Fatal Error: An error caused by process due to a bug in program for example, executing an illegal instruction, referring non-existing memory or dividing by zero.
4. Killed by another Process: A process executes a system call telling the Operating Systems to terminate some other process. In UNIX, this call is `kill`. In some systems when a process kills all processes it created are killed as well (UNIX does not work this way).

Threads

Threads are same as *Processes*, to do more than one thing at a time. Like processes, threads also to run concurrently.

[Thread Definition] A thread is a basic unit of CPU utilization. We may have one or more thread within a same process. Threads are a finer-grained unit of execution than processes. That is the reason sometimes they are called *Lightweight Processes*.

When you invoke a program, OS creates a new process. Process may create a single thread, which runs the program sequentially. That thread can create additional threads to perform different tasks concurrently.

For example, when we run a word-processor program, a single thread is executing to perform all task. This single thread allows the process to perform only one task at one time. The user

can not simultaneously type in characters and run spell checker within the same process. But Modern word processor like Microsoft Word allows user to type at the same time spellchecker and autorecovery thread(Which saves text at periodical interval) is also working. Multiple threads in same Microsoft Word process made it possible.

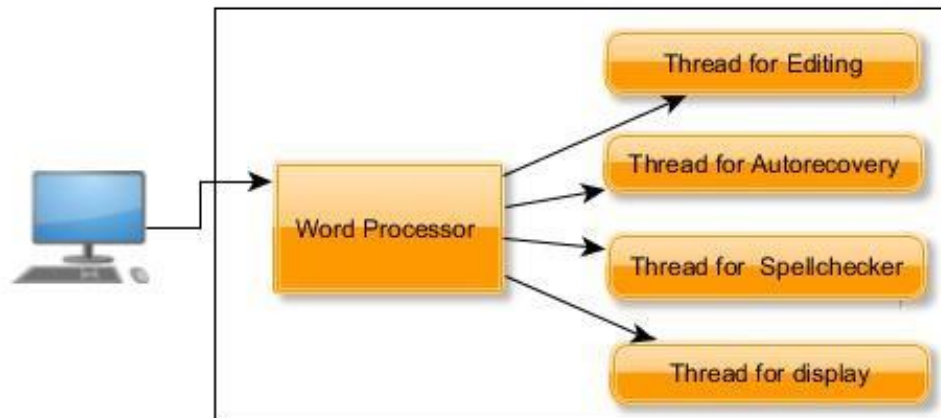


Figure 2.6: Thread Example: Word Processor Application process may contain different threads

Benefits

These are few *reasons* or *benefits* to use threads over process:

Concurrency In many applications multiple activities are going on concurrently. By decomposing such an application into multiple sequential threads that run in virtually parallel manner, the programming model becomes simpler.

Responsiveness Multi threading may allow a program to interact even if part of it is blocked or doing processing. For example, a multi-threaded web browser allow user to open new tab while other tab is loading.

Faster They are lighter weight than processes. they can be created and destroyed faster and easily than processes. In many systems, creating a thread 10-100 times faster than creating a process.

Performance Threads may not give performance gain when all of them are CPU bound. Most of the time a task is mix of CPU burst and I/O burst. Threads can overlap these activities, thus speeding up the application.

Use of Multiple CPUs Threads are useful for Multicore / Multiprocessor systems.

In those systems, Each execution unit can be assigned a single thread.

In this way we can utilize multiple processor.

Difference between thread and process

Thread generally contains less information than process. It consists of a thread ID, a program counter, a register set, and a stack. It shares code section, data section, and other operating-system resources, such as open files and signals with other threads in same process. We summarize here difference between Threads and Processes:

1. Processes are typically independent, while threads exist as subsets of a process.
2. Processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources.
3. Processes have separate address spaces, whereas threads share their address space.

4. Processes interact only through system-provided inter-process communication mechanisms.
5. Context switching between threads in the same process is typically faster than context switching between processes.

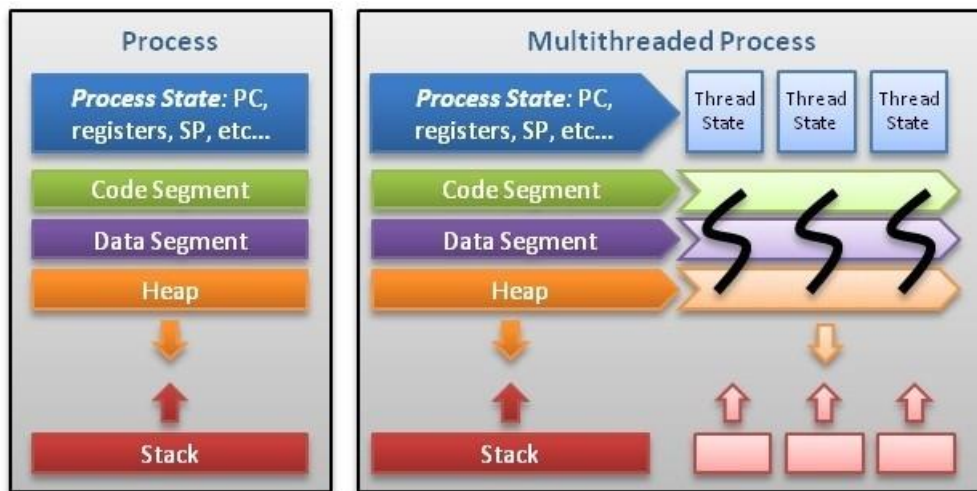


Figure 2.7: Thread vs Process

Types of Threads

Threads are useful in writing interactive application and improving CPU utilization. Threads are supported at two levels : kernel threads and user threads. User threads are supported above the kernel and are managed without kernel support. User threads runs in user space. Kernel threads are supported and managed directly by the operating system. Kernel threads runs in kernel space. Virtually all operating systems for example Windows XP, Linux, Mac OS X, Solaris, and UNIX support kernel threads. A thread library provides the programmer an API for creating and managing threads.

Three main thread libraries are in use today:

1. POSIX Pthreads
2. Win32
3. Java.

Pthreads, the threads extension of the POSIX standard, supports both user and kernel level threads. Pthreads library is available in UNIX and LINUX OS. The Win32 thread library is a kernel-level library available on Windows systems. The Java thread API allows thread creation and management directly in Java programs.

User Level Threads

The first approach is to provide a library entirely in user space with no kernel support. User-Level threads can be created easily than process because, much less state to allocate and initialize. They are implemented at user level. User-Level threads are managed entirely by the run-time system (user-level library). The kernel knows nothing about user-level threads and manages them as if they were single-threaded processes. User-Level threads are small and fast, each thread is represented by a PC, register, stack, and small thread control block. Creating a new thread, switching between threads, and synchronizing threads are done via procedure call. i.e no kernel involvement. User-Level threads are hundred times faster than Kernel-Level threads.

Benefits:

1. A user-level threads package can be implemented on an Operating System that does not support threads.
2. User threads are easy and fast to create.
3. Communication
4. Thread switching is not much more expensive than a procedure call.

Disadvantages:

1. User-Level threads are invisible to the OS they are not well integrated with the OS. As a result, OS can make poor decisions like scheduling a process with idle threads. OS may blocking a process whose thread initiated an I/O even though the process has other threads that can run. OS may stop a process with a thread holding a lock. Solving this requires communication between kernel and user-level thread manager.
2. There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.
3. User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

Examples: Java implements threads as User level threads.

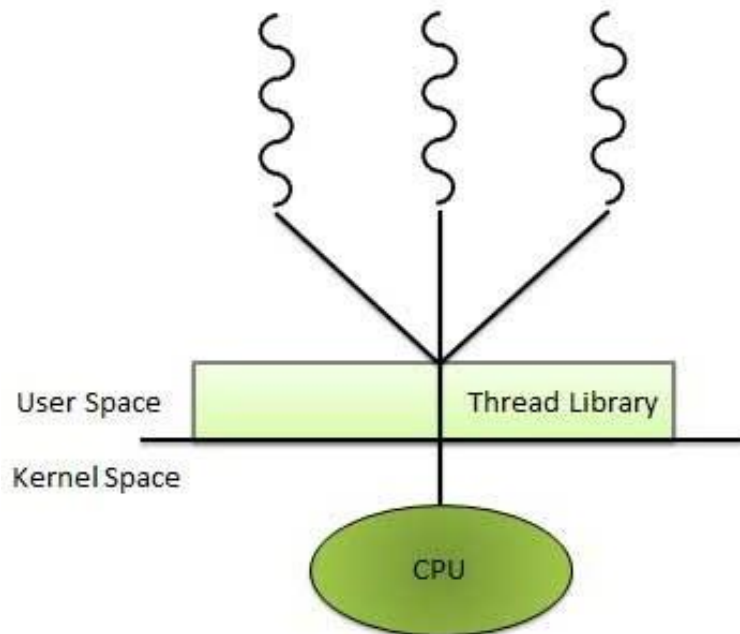


Figure 2.8: User Level Threads

Kernel Level Threads

Operating system manages threads. All thread operations are implemented in the kernel. The OS schedules all threads in the system. OS managed threads are called kernel-level threads. They sometimes are called *Light Weight Processes*. In this method, the kernel knows about threads. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.

Benefits:

1. Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
2. Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

1. The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.
2. Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity.

Examples:

Windows NT implement Kernel Level Threads and Solaris as Lightweight processes(LWP).

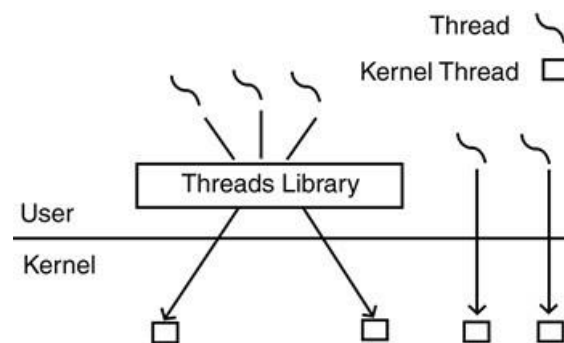


Figure 2.9: Kernel Level Thread

Scheduling

Scheduling Queues

Scheduler's Types

Scheduling Criteria

Preemptive Vs Nonpreemptive Scheduling

Scheduling Algorithm

First Come First-Served Scheduling
(FCFS)
Shortest-Job-First Scheduling (SJF)
Preemptive SJF or Shortest Remaining
Time First (SRTF)
Priority Scheduling
Round Robin Scheduling
Multilevel Queue Scheduling

Multiprocessor Scheduling Algorithm

Symmetric MultiProcessor SMP
Asymmetric multiprocessing

Scheduling Algorithm Evaluation

Computer frequently switches among multiple processes or threads in case of Multiprogramming. There may be two or more processes are staying in the ready state. Scheduler, which is the part of OS is responsible in making decision that which process to run next. Scheduler uses various algorithm for it, which are called as scheduling algorithms.

Scheduling Queues

Data structures play an important role in management of processes. OS may use more than one data structure in the management of processes.

It may maintain following scheduling queues:

Ready Queue All ready to run processes are contained in queue

Waiting Queue OS maintains separate queues for blocked processes. It may even have a separate queue for each of the likely events (including completion of IO).

Job Queue When the process enters into the system, it is put into a job queue. This queue consists of all processes in the system.

Device Queue It is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

Scheduler's Types

Scheduler's main task is to select the jobs to be submitted into the system and to decide which process to run.

Schedulers are of three types:

1. **Long Term Scheduler** It is also called *job scheduler*. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also 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. On some systems, the long term scheduler may not be available or minima

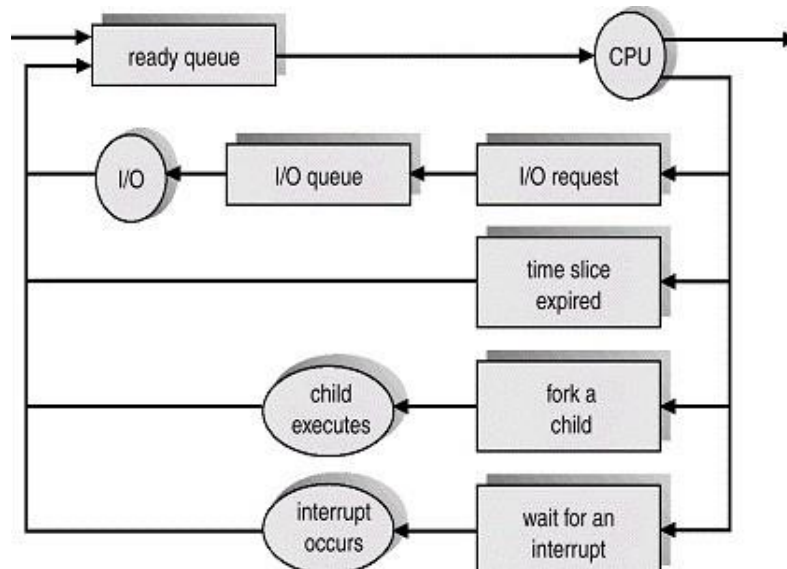


Figure 3.1: Scheduling Queue

Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready, then there is use of long term scheduler.

2. **Short Term Scheduler** It is also called *CPU scheduler*. CPU scheduler selects process among the processes that are ready to execute and allocates CPU to one of them. Short term scheduler also known as *dispatcher*, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.
3. **Medium Term Scheduler** Medium term scheduling is part of the *swapping*. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.

[Swapping:] Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Comparison between Scheduler

S.N.	Long Term Scheduler	Short Term Scheduler	Medium Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Scheduling Criteria

There are various criteria for scheduling. Some criterion's are important in some system. For example, In server OS improving CPU utilization and system performance is objective. While in Timesharing or PC OS user response is important. Many objectives must be considered in the design of a scheduling discipline. In particular, a scheduler should consider fairness, efficiency, response time, turnaround time, throughput.

There are also some goals that are desirable in all systems.

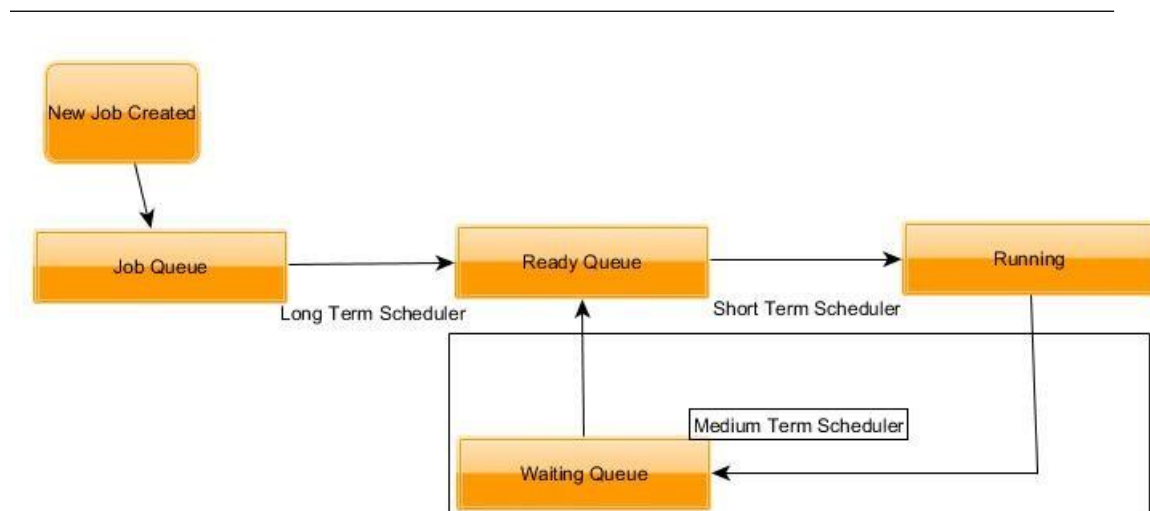


Figure 3.2: Types of Schedulers

Fairness Fairness is important under all circumstances. A scheduler makes sure that each process gets its fair share of the CPU. It should not happen that any process will wait very long time (indefinite postponement). Note that giving equivalent or equal time is not fair. Think of safety control and payroll at a nuclear plant.

Policy Enforcement The scheduler has to make sure that system's policy is enforced. For example, if the local policy is safety then the safety control processes must be given highest priority. It is always scheduled before other processes.

Efficiency Scheduler should keep the system (or in particular CPU) busy always or most of the time. If the CPU and all the Input/Output devices can be kept running all the time, more work gets done per second.

Response Time A scheduler should minimize the response time for interactive user. The response time is the time from the submission of a request until the first response is produced.

Turnaround Time 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. It is important to reduce turn around time in batch system. A scheduler should minimize the time batch users must wait for an output.

Throughput Throughput is the number of processes that are completed per time unit. A scheduler should maximize the throughput. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.

Waiting Time Waiting time is total time process spent waiting in the ready queue.

It is desirable to *maximize Efficiency and throughput* and to *minimize turnaround time, waiting time, and response time*. Some of these goals are contradictory. In some cases, we may focus on one criteria more than other. For example, to guarantee that all users get good service, we may want to minimize response time but it may not give you efficiency. While if scheduler gives importance on efficiency, it may not be able to provide quick response to user.

Preemptive Vs Non preemptive Scheduling

The Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

1. Nonpreemptive Scheduling

A scheduling discipline is nonpreemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process.

Following are some characteristics of nonpreemptive scheduling:

- (a) In nonpreemptive scheduling, a scheduler executes jobs in the following two situations:
 - 1. When a process switches from running state to the waiting state.
 - 2. When a process terminates.
- (b) In nonpreemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
- (c) In nonpreemptive system, response times are more predictable because incoming high priority jobs can not displace waiting jobs.

2. Preemptive Scheduling

A scheduling discipline is preemptive if, scheduler is able to take CPU from process before it leaves CPU voluntary. In preemptive scheduling, processes can go from running to be temporarily suspended while in non preemptive scheduling processes run to completion method.

Scheduling Algorithm

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.

First-Come, First-Served Scheduling (FCFS)

Simplest CPU-scheduling algorithm is the first-come, first-served(FCFS) scheduling algorithm. We can take an analogy, in railway reservation queue person who comes first will get his ticket first. In this algorithm, 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.

Consider the following set of processes that arrive in order(P1,P2,P3,P4), with the length of the CPU burst given in milliseconds:

Process	CPU Burst Time
P1	20
P2	6
P3	4
P4	2

For this example processes arrived in the order P1,P2,P3,P4. Scheduler which uses FCFS allocate CPU is same order.

Following Gantt chart shows execution of processes:

P1	P2	P3	P4	
0	20	26	30	32

The waiting time (WT) for processes will be:

$$WT(P1) = 0$$

$$WT(P2) = 20$$

$$WT(P3) = 26$$

$$WT(P4) = 30$$

$$AverageWT = 0 + 20 + 26 + 30/4$$

The turnaround time(TA) for processes will be:

$$TA(P1) = 20$$

$$TA(P2) = 26$$

$$TA(P3) = 30$$

$$TA(P4) = 32$$

$$AverageTA = 20 + 26 + 30 + 32/4$$

The benefit of this algorithm is that it is easy to understand and easy to implement. It is also fair in the same sense that process requested CPU first will get CPU first.

If same processes arrive in order P4,P3,P2,P1 than the waiting time for processes will be:

P4	P3	P2	P1
0	2	6	12
			32

$$WT(P1) = 12$$

$$WT(P2) = 6$$

$$WT(P3) = 2$$

$$WT(P4) = 0$$

$$AverageWT = 12 + 6 + 2 + 0/4$$

The turnaround time(TA) for processes will be:

$$TA(P1) = 2$$

$$TA(P2) = 6$$

$$TA(P3) = 12$$

$$TA(P4) = 32$$

$$AverageTA = 2 + 6 + 12 + 30/4$$

The average waiting time under the FCFS is generally very long. It depends upon the arrival order of process. We can take an example, assume we have one CPU-bound process (large CPU burst) and many I/O-bound processes (small CPU burst). If the CPU-bound process arrives first. Scheduler will allocate CPU First to that process. It 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. 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 will have to wait until the CPU-bound process completes its CPU burst. This effect is known as *convoy effect* in which all the other processes wait for the one big process to leave CPU. This effect results in lower CPU and device utilization than allowing shorter processes to complete execution. The FCFS scheduling algorithm is *nonpreemptive*. The FCFS algorithm is not suitable for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

Shortest-Job-First Scheduling(SJF)

A different approach to CPU scheduling is the shortest-job-first (SJF) or Shortest Process First(SPF) scheduling algorithm. This algorithm select shortest process for scheduling. 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. This

scheduling method can also be called 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.

As an example of SJF scheduling, consider the following set of processes:

Process	CPU Burst Time
P1	8
P2	4
P3	4
P4	4

For this example processes arrived in the order P1,P2,P3,P4. Scheduler which uses SJF allocate CPU to shortest process. In this case P2, P3,P4 are having next burst of 4 than scheduler pick P2 which comes before P3,P4. After P2, Scheduler choose next process in order P3, P4, P1 Following Gantt chart shows execution of processes:

The waiting time(WT) for processes will be:

$$WT(P1) = 12$$

$$WT(P2) = 0$$

$$WT(P3) = 4$$

$$WT(P4) = 8$$

$$AverageWT = 12 + 0 + 4 + 8/4$$

The turnaround time(TA) for processes will be:

$$TA(P1) = 20$$

$$TA(P2) = 4$$

$$TA(P3) = 8$$

$$TA(P4) = 12$$

$$AverageTA = 20 + 4 + 8 + 12/4$$

The SJF scheduling algorithm is optimal algorithm. It always gives minimum average waiting time and turn around time for a given set of processes. By completing a short process before a long, we can decrease the overall waiting time.

The real difficulty with the SJF algorithm is knowing the length of the next CPU request. SJF scheduling is used frequently in long-term scheduling. 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. The user can give short time to get faster response, which may lead to poor performance. SJF algorithm cannot be implemented at the level of short-term CPU scheduling. Because there is no way to know the length of the next CPU burst.

One approach is to use approximate SJF scheduling. We may not know the length of the next CPU burst. But we can use previous CPU burst length to predict value of next CPU burst. Thus, by computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst. The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let t_n be the length of the n th CPU burst and T_{n+1} is predicted value of $n+1$ burst then

$T_{n+1} = at_n + (1 - a)T_n$ Where a is a constant and $0 \leq a \leq 1$ holds. This formula is exponential average formula. In which a controls weightage of historical and recent value of CPU burst. t_n is the recent value of CPU burst and T_n is our approximate CPU burst calculated on basis of past values of CPU burst.

If $a = 0$, then $T_{n+1} = T_n$ and recent value of CPU burst has no effect. If $a = 1$, then $T_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, $a = 1/2$, so recent history and past history are equally weighted.

We can take an example: If process last CPU burst is 2, $a = 0.5$ and assuming last predicted value of $T_n = 0$ then new estimated value of next CPU Burst is: $T_{n+1} = 0.5 * 2 + (1 - 0.5) * 0 = 1$ in

same way T_{n+2} can be calculated (if $t_{n+1} = 4$): $T_{n+2} = .5 * 4 + (1 - 0.5) * 1 = 2.5$

Preemptive SJF or Shortest Remaining Time First (SRTF)

In a nonpreemptive SJF algorithm, scheduler will allow the currently running process to finish its CPU burst, even if process with shorter CPU burst comes in ready queue. Whereas SRTF, will preempt the currently executing process if process with shorter burst request for execution. Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling. We can take an following example

Process	CPU Burst Time	Arrival Time
P1	8	0
P2	4	1
P3	2	2
P4	4	3

Preemptive SJF scheduler schedule in following way:

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. Process P3 arrive at time 2. The remaining time for process P1 (7 milliseconds) and P2 is (3 milliseconds) are larger than the time required by process P3 (2 milliseconds), so process P2 is preempted, and process P3 is scheduled. At time 3, process P4 arrived. The remaining time (1) for P3 is shortest among all. So it will complete its execution first, after that P2, P4, P1 completes their execution in order.

	P1	P2	P3	P2	P4	P1
0	1	2	4	7	11	18

The waiting time (WT) for processes will be:

$$WT(P1) = 11 - 1 - 0 = 10$$

$$WT(P2) = 4 - 1 - 1 = 2$$

$$WT(P3) = 2 - 2 = 0$$

$$WT(P4) = 7 - 3 = 4$$

Average WT = $10 + 2 + 0 + 4 / 4$ The turnaround time (TA) for processes will be:

$$TA(P1) = 18 - 0 = 18$$

$$TA(P2) = 7 - 1 = 6$$

$$TA(P3) = 4 - 2 = 2$$

$$TA(P4) = 11 - 3 = 8$$

Average TA = $18 + 6 + 2 + 8 / 4$ While for same set of processes Nonpreemptive SJF scheduling would result in an average waiting time of $0 + 6 + 9 + 11 / 4$ milliseconds.

Priority Scheduling

In priority scheduling algorithm, scheduler will choose process with 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. 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. Here, we assume that low numbers

represent high priority.

Consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2, P3, P4, P5, with the length of the CPU burst given in milliseconds: The scheduler choose Highest

Process	CPU Burst Time	Priority
P1	8	3
P2	2	1
P3	3	4
P4	4	4
P5	5	2

priority process P2 first, then choose processes in order P5, P1, P3, P4. The Gantt chart for this algorithm given as:

	P2	P5	P1	P3	P4
0	2	7	15	18	22

The waiting time(WT) for processes will be:

$$WT(P1) = 7$$

$$WT(P2) = 0$$

$$WT(P3) = 15$$

$$WT(P4) = 18$$

$$WT(P5) = 2$$

$$AverageWT = 7 + 0 + 15 + 18 + 2 / 5$$
 The turnaround time(TA) for processes will be:

$$TA(P1) = 15$$

$$TA(P2) = 2$$

$$TA(P3) = 18$$

$$TA(P4) = 22$$

$$TA(P5) = 7$$


$$AverageTA = 15 + 2 + 18 + 22 + 7 / 5$$

Few characteristics of Priority Scheduling:

1. Priorities can be defined either internally or externally.

Internally defined priorities are defined by Operating system. For example OS can calculate priority on basis of 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 OS. It depends upon the importance of the process, the type of user initiated process.

2. Priority scheduling can be either pre-emptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A pre-emptive 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 nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
3. 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 priority scheduling algorithm are used with combination of round-robin scheduling. We can define priority classes based on priority. If there are runnable processes in priority class 4, just run each one for one quantum, round-robin fashion. If priority class 4 is empty, then run the class 3 processes round robin. If classes 4 and 3 are both empty, then run class 2 round robin, and so on. If processes belongs to lower priority classes, then they may be starved.because it may happen that high priority process comes one after another and low priority process is waiting long for processor

[Aging] In aging priorities of jobs increase the longer they wait. Under this scheme scheduler increase priority of a low-priority job with time. They moved up in high priority queue, where they have better chance to avail CPU.

Round Robin Scheduling

In Round Robin(RR) Scheduling,Each process is provided a fix time to execute called quantum. Once a process is executed for given time period. Process is preempted and other process executes for given time period. Context switching is used to save states of preempted processes. RR scheduling involves extensive overhead, especially with a small time unit. RR scheduling provides balanced throughput between FCFS and SJF. In RR scheduling,shorter jobs are completed faster than in FCFS and longer processes are completed faster than in SJF. RR scheduling provides good average response time. In this,waiting time is dependent on number of processes, and not average process length. System using RR may suffer from high waiting times and may not be able to met deadlines . Starvation can never occur, since no priority is given. It is similar to FCFS, order of time unit allocation is based upon process arrival time. If size of time quantum is too large than it may behave like FCFS. That is the reason it may called as *Preemptive FCFS*. We can take following example to understand Round Robin Scheduling : For above set of processes,

Process	Burst Time
P1	24
P2	4
P3	4

RR Scheduler choose P1 first and allocate it CPU for time quantum =4. If process voluntary leave CPU before time quantum then it will allocate CPU to other process. In this case P1 does not leave CPU voluntary, that is the reason CPU will preempt P1 and allocate CPU to next process P2 for 4 ms. After P2 completes scheduler picks next process in ready queue which is P3. After completion of P3 scheduler picks P1 and so on. The waiting time(WT) for processes will be:

$$WT(P1) = 8$$

$$WT(P2) = 4$$

$$WT(P3) = 8$$

$AverageWT = 8 + 4 + 0/3$ The turnaround time(TA) for processes will be:

$$TA(P1) = 32$$

$$TA(P2) = 8$$

$$TA(P3) = 12$$

$$AverageTA = 32 + 8 + 12/3$$

Gantt Chart for time quantum=4 :

- R The selection of time quantum(length of time quantum) is important for RR Scheduling. If we choose small time quantum, we will get better response but may suffers from large

	P1	P2	P3	P1	P1	P1	P1	P1
0	4	8	12	16	20	24	28	32

number of context switch. If we take large time quantum, our scheduling behaves like FCFS which is having problem of higher waiting time.

What will happen if we increase time quantum for above example. The waiting time(WT) for processes will be:

$$WT(P1) = 0$$

$$WT(P2) = 8$$

$$WT(P3) = 12$$

$AverageWT = 0 + 8 + 12/3$ The turnaround time(TA) for processes will be:

$$TA(P1) = 32$$

$$TA(P2) = 12$$

$$TA(P3) = 16$$

$$AverageTA = 32 + 12 + 16/3$$

Gantt chart for time quantum=8 :


	P1	P2	P3	P1	P1	P1
0	8	12	16	24	32	

The characteristics of RR Scheduling:

1. The main advantage of round robin algorithm over FCFS is that it is *starvation free*. Every process will be executed by CPU for fixed interval of time (which is set as time slice). So in this way no process left waiting for its turn to be executed by the CPU .
2. RR algorithm is simple and easy to implement .
3. The length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS.

Multilevel Queue Scheduling

In this scheduling processes grouped like foreground(interactive process) ,background(batch process),etc.These two types of processes have different response-time requirements. They should be scheduled differently. A multilevel queue scheduling algorithm partitions the ready queue into multiple queues. 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 can have its own scheduling algorithms.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.

 Note that under this algorithm jobs cannot switch from queue to queue. Once they are assigned a queue, they remain in same queue until they finish.

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

Upper queue has greater priority over its lower-priority queues. No process in the batch queue

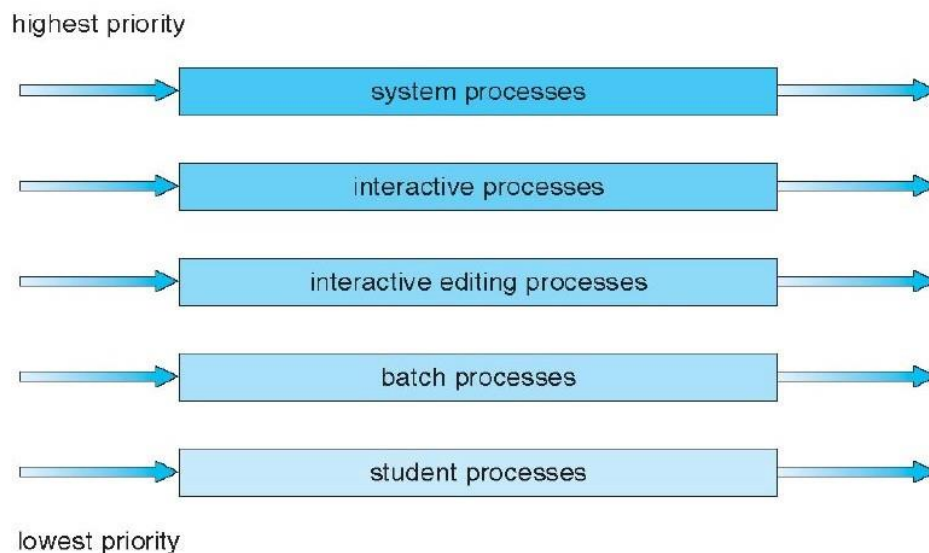


Figure 3.3: Multilevel Queue Scheduling

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 option is to give fixed time-slice to 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. Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. Processes do not able to change their queues(foreground or background nature). This setup has the advantage of low scheduling overhead, but it is inflexible.

[Multilevel Feedback Scheduling:] It is one special case of Multilevel queue scheduling where processes can change their queues. The multilevel feedback-queue scheduling algorithm, allows a process to move between queues. It may separate processes based on length of their CPU bursts.

If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. The technique of aging can also be implemented in this scheduling. Process waiting in a lower-priority queue from long time may be moved to a higher-priority queue. We can summarize scheduling goals and algorithms suitable for different systems:

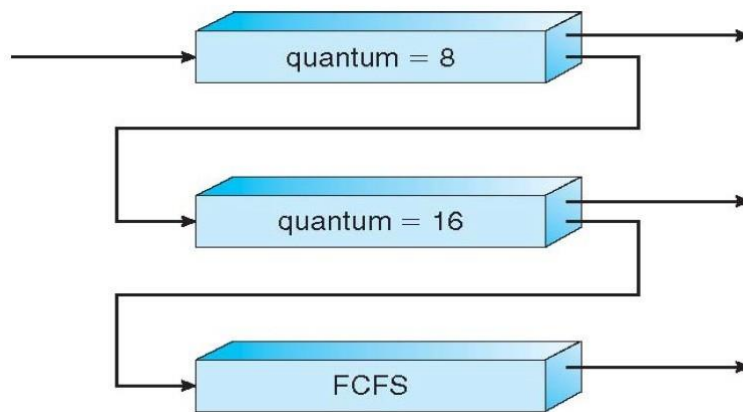


Figure 3.4: Multilevel Feedback Queue Scheduling

System	Goal	Scheduling Algorithms
Batch	Throughput	FCFS
	Turn around time	Shortest Job First
	CPU utilization	Shortest Remaining Time First
Interactive	Response Time	Round Robin
	Proportionality	Priority
Real Time	Meeting Deadline	Guaranteed Scheduling
	Predictability	

Multiprocessor Scheduling Algorithm

Multiprocessor systems are increasingly commonplace, and have found their way into desktop machines, laptops, and even mobile devices. In this system, Multiple CPUs are available, load sharing is possible. Multiprocessor Scheduling can be divided in following two types:

Symmetric MultiProcessor SMP

The most common multiprocessor system in use today is the Symmetric MultiProcessor (SMP). All of the CPUs in an SMP system are identical. All processes put into a common ready queue, or each processor may have its own private queue of ready processes. Whenever a CPU needs a process to run, it takes the next task from the ready list. The scheduling queue must be accessed in a critical section. Busy waiting is usually used.

Some consideration for SMP scheduling:

1. Selection of the next task is not as important. With multiple CPUs, it is not as likely that a short task will wait for a long task to complete.
2. Any task can run on any CPU thereby allowing load balancing.
3. Tasks should stay with a single CPU to take advantage of cache loading.
4. It can use *Gang scheduling*.

[Gang Scheduling:] It schedule all of the threads of a process together.

In an SMP, this isn't much of a problem since any CPU can execute any thread. In systems with distributed memory (each CPU has its own memory and cannot access the RAM of another CPU), load sharing is a major consideration. This is often more of an application concern than an OS concern.

Asymmetric multiprocessing

In this approach, *master server* handles all scheduling decisions, I/O processing, and other system activities. The other processors execute only user code. This asymmetric multiprocessing is simple because only one processor accesses the system data structures, reducing the need for data sharing. In an asymmetric multiprocessing system, not all CPUs are treated equally; for example, a system might only allow (either at the hardware or operating system level) one CPU to execute operating system code or might only allow one CPU to perform I/O operations.

Scheduling Algorithm Evaluation

How can we select a CPU scheduling algorithm for a particular system. We can evaluate scheduling algorithms by establishing criteria. The criteria may be like Maximizing CPU utilization, Guaranteed response time, Maximizing throughput, etc or combination of this. There are many scheduling algorithms. We can evaluate CPU algorithm based on criteria by following methods:

1. **Deterministic modeling** : We can take a particular predetermined workload and define the performance of each algorithm for the workload. For example Consider the FCFS, SJF, and RR(time quantum=10) scheduling algorithms for set of processes. which will give the min avg. WT? We have done deterministic modeling by using Gantt chart in scheduling algorithm examples. Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms. But, it requires exact numbers for input, and its answers apply only to those cases.
2. **Queuing models** : It uses probabilistic approach to model arrival of processes, and CPU and I/O bursts. It Computes average throughput, utilization, waiting time, etc. In it, computer system described as network of servers, each with queue of waiting processes. This area of study is called queueing-network analysis. Queueing analysis can be useful in comparing scheduling algorithms, but it also has limitations. The classes of algorithms and distributions that can be handled with this model are limited. The mathematics of complicated algorithms and distributions can be difficult to work with. Queueing models are often only approximations of real systems, and the accuracy of the computed results may be questionable.
3. **Simulations** : Simulations are more accurate than queuing model. They use Programmed model of computer system. They Gather statistics indicating algorithm performance. Simulation uses Random number generator according to probabilities Distributions defined mathematically or empirically to provide input. Simulations can be expensive, often requiring hours of computer time. A more detailed simulation provides more accurate results, but it also requires more computer time. Finally, the design, coding, and debugging of the simulator can be a major task.
4. **Implementation** : Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions. The major difficulty with this approach is the high cost. Another difficulty is that the environment in which the algorithm is used will change.

Real Time Scheduling

Priority based scheduling enables us to give better service to certain processes. In our discussion of multi-queue scheduling, priority was adjusted based on whether a task was more interactive or compute intensive. But most schedulers enable us to give any process any desired priority. Isn't that good enough?

Priority scheduling is inherently a *best effort* approach. If our task is competing with other high priority tasks, it may not get as much time as it requires. Sometimes best effort isn't good enough:

- During reentry, the space shuttle is aerodynamically unstable. It is not actually being kept under control by the quick reflexes of the well-trained pilots, but rather by guidance computers that are collecting attitude and acceleration input and adjusting numerous spoilers hundreds of times per second.
- Scientific and military satellites may receive precious and irreplaceable sensor data at extremely high speeds. If it takes us too long to receive, process, and store one data frame, the next data frame may be lost.
- More mundanely, but also important, many manufacturing processes are run by computers nowadays. An assembly line needs to move at a particular speed, with each step being performed at a particular time. Performing the action too late results in a flawed or useless product.
- Even more commonly, playing media, like video or audio, has real time requirements. Sound must be produced at a certain rate and frames must be displayed frequently enough or the media becomes uncomfortable to deal with.

There are many computer controlled applications where delays in critical processing can have undesirable, or even disastrous consequences.

What are Real-Time Systems

A real-time system is one whose correctness depends on timing as well as functionality.

When we discussed more traditional scheduling algorithms, the metrics we looked at were *turn-around time* (or throughput), *fairness*, and mean *response time*. But real-time systems have very different requirements, characterized by different metrics:

Terminology

Hard real-time task - a process that has deadlines to meet or the results are useless

Soft real-time task - a process that has deadlines but they're not mandatory

Periodic task - a process that has to carry out its task in regular time intervals

Aperiodic task - a process that has a constraint on the start or the stop time; it may be asynchronous in nature

- *timeliness* ... how closely does it meet its timing requirements (e.g. ms/day of accumulated tardiness)
- *predictability* ... how much deviation is there in delivered timeliness

And we introduce a few new concepts:

- *feasibility* ... whether or not it is possible to meet the requirements for a particular task set
- *hard real-time* ... there are strong requirements that specified tasks be run at specified intervals (or within a specified response time). Failure to meet this requirement (perhaps by as little as a fraction of a micro-second) may result in system failure.
- *soft real-time* ... we may want to provide very good (e.g. microseconds) response time, the only consequences of missing a deadline are degraded performance or recoverable failures.

It sounds like real-time scheduling is more critical and difficult than traditional time-sharing, and in many ways it is. But real-time systems may have a few characteristics that make scheduling easier:

- We may actually know how long each task will take to run. This enables much more intelligent scheduling.
- *Starvation* (of low priority tasks) may be acceptable. The space shuttle absolutely must sense attitude and acceleration and adjust spoiler positions once per millisecond. But it probably doesn't matter if we update the navigational display once per millisecond or once every ten seconds. Telemetry transmission is probably somewhere in-between. Understanding the relative criticality of each task gives us the freedom to intelligently shed less critical work in times of high demand.
- The work-load may be relatively fixed. Normally high utilization implies long queuing delays, as bursty traffic creates long lines. But if the incoming traffic rate is relatively constant, it is possible to simultaneously achieve high utilization and good response time.

Real-Time Scheduling Algorithms

In the simplest real-time systems, where the tasks and their execution times are all known, there might not even be a scheduler. One task might simply call (or yield to) the next. This model makes a great deal of sense in a system where the tasks form a producer/consumer pipeline (e.g. MPEG frame receipt, protocol decoding, image decompression, display).

In more complex real-time system, with a larger (but still fixed) number of tasks that do not function in a strictly pipeline fashion, it may be possible to do *static* scheduling. Based on the list of tasks to be run, and the expected completion time for each, we can define (at design or build time) a fixed schedule that will ensure timely execution of all tasks.

But for many real-time systems, the work-load changes from moment to moment, based on external events. These require *dynamic* scheduling. For *dynamic* scheduling algorithms, there are two key questions:

1. how they choose the next (ready) task to run
 - shortest job first
 - static priority ... highest priority ready task
 - soonest start-time deadline first (ASAP)
 - soonest completion-time deadline first (slack time)
2. how they handle overload (infeasible requirements)
 - best effort
 - periodicity adjustments ... run lower priority tasks less often.
 - work shedding ... stop running lower priority tasks entirely.

Preemption may also be a different issue in real-time systems. In ordinary time-sharing, preemption is a means of improving mean response time by breaking up the execution of long-running, compute-intensive tasks. A second advantage of preemptive scheduling, particularly important in a general purpose timesharing system, is that it prevents a buggy (infinite loop) program from taking over the CPU. The trade-off, between improved response time and increased overhead (for the added context switches), almost always favors preemptive scheduling. This may not be true for real-time systems:

- preempting a running task will almost surely cause it to miss its completion deadline.
- since we so often know what the expected execution time for a task will be, we can schedule accordingly and should have little need for preemption.
- embedded and real-time systems run fewer and simpler tasks than general purpose time systems, and the code is often much better tested ... so infinite loop bugs are extremely rare.

For the least demanding real time tasks, a sufficiently lightly loaded system might be reasonably successful in meeting its deadlines. However, this is achieved simply because the frequency at which the task is run happens to be high enough to meet its real time requirements, not because the scheduler is aware of such requirements. A lightly loaded machine running a traditional scheduler can often display a video to a user's satisfaction, not

because the scheduler "knows" that a frame must be rendered by a certain deadline, but simply because the machine has enough cycles and a low enough work load to render the frame before the deadline has arrived.

Real-Time and Linux

Linux was not designed to be an embedded or real-time operating system, but many tasks that were once-considered embedded applications now require the capabilities (e.g. file systems, network protocols) of a general purpose operating system. As these requirements have increased and processors have gotten faster, increasingly many embedded and real-time applications have moved to Linux.

To support these applications Linux now supports a real-time scheduler, which can be enabled with `sched_setscheduler`. This real-time scheduler does not provide quite the same level of response-time guarantees that more traditional Real-Time-OSs do, but they are adequate for many soft real-time applications.

Windows

Conventional wisdom states that Windows is not well suited for real time needs, offering no native real time scheduler and too many ways in which desired real time deadlines might be missed. Windows favors general purpose throughput over meeting deadlines, as a rule. With sufficiently low load, a Windows system may, nonetheless, provide fast enough service for some soft real time requirements, such as playing music or video. One should be careful in relying on Windows for critical real time operations, however, as it is not designed for that purpose.