Chapter 2: Process Management

2.1 Process

A process is defined as an entity which represents the basic unit of work to be implemented in the system i.e. a process is a program in execution. The execution of a process must progress in a sequential fashion. In general, a process will need certain resources such as the CPU time, memory, files, I/O devices and etc. to accomplish its task.

Operation on Process

- Process Creation
- Destroy of a process
- Run a process
- Change a process priority
- Get process information
- Set process information

Process Creation

It's a job of OS to create a process. There are four ways achieving it:

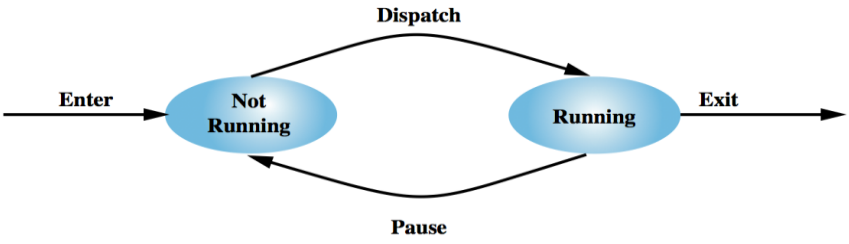
1. For a batch environment a process is created in response to submission of a job.
2. In Interactive environment, a process is created when a new user attempt to log on.
3. The OS can create process to perform functions on the behalf a user program.
4. A number of process can be generated from the main process. For the purpose of modularity or to exploit parallelism a user can create numbers of process.

Process Termination

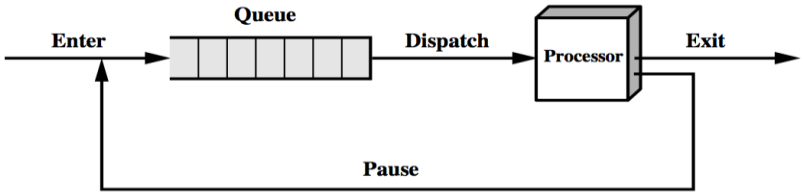
A process terminates when it finishes executing its last statement. Its resources are returned to the system, it is purged from any system lists or tables, and its process control block (PCB) is erased. The new process terminates the existing process, usually due to following reasons:

- **Normal Exit (Voluntary):** Most processes terminate because they have done their job.
- **Error Exist(Voluntary):** When process discovers a fatal error. For example, a user tries to compile a program that does not exist.
- **Fatal Error (Involuntary):** 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.
- **Killed by another Process (Industrial)**: 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).

Process State (Two state model)



(a) State transition diagram



(b) Queuing diagram

In this model, we consider two main states of the process. These two states are

State 1: Process is Running on CPU
State 2: Process is Not Running on CPU

New: First of all, when a new process is created, then it is in Not Running State. Suppose a new process P2 is created then P2 is in NOT Running State.

CPU: When CPU becomes free, Dispatcher gives control of the CPU to P2 that is in NOT Running state and waiting in a queue.

Dispatcher: Dispatcher is a program that gives control of the CPU to the process selected by the CPU scheduler. Suppose dispatcher allow P2 to execute on CPU.

Running: When dispatcher allows P2 to execute on CPU then P2 starts its execution. Here we can say that P2 is in running state.

Now, if any process with high priority wants to execute on CPU, Suppose P3 with high priority, then P2 should be the pause or we can say that P2 will be in waiting state and P3 will be in running state.
Now, when P3 terminates then P2 again allows the dispatcher to execute on CPU

Process State (Five state model)

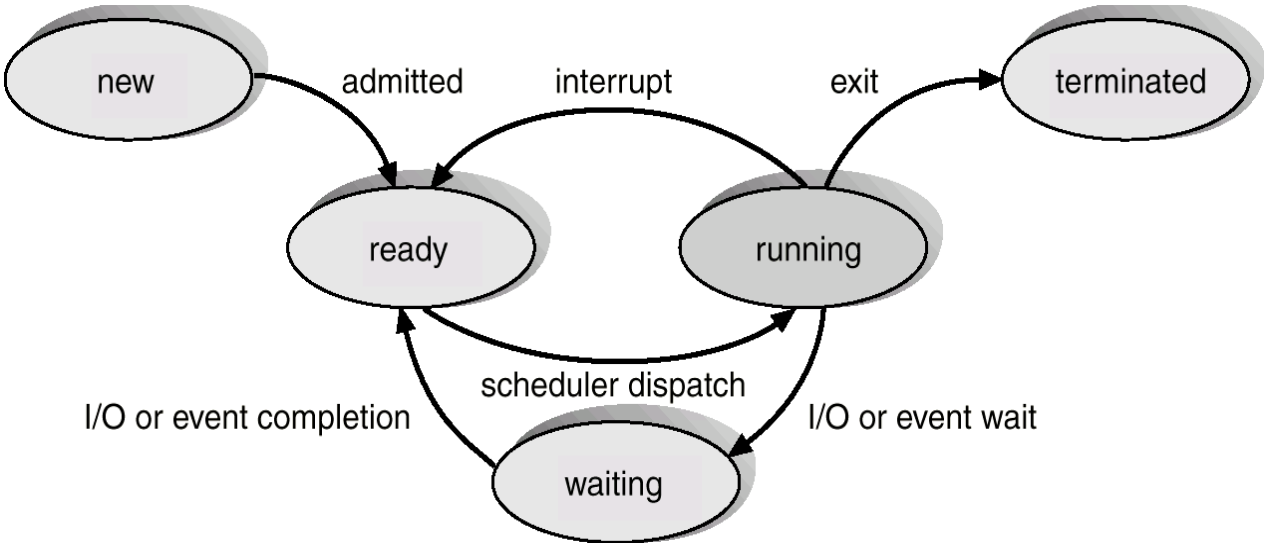


Fig: Five States Process Model

1. **New:** A process that has just been created but has not yet been admitted to the pool of executable processes by the OS.
2. **Ready:** Process that is prepared to execute when given the opportunity. That is, they are not waiting on anything except the CPU availability.
3. **Running:** the process that is currently being executed.
4. **Blocked:** A process that cannot execute until some event occurs, such as the completion of an I/O operation.
5. **Exit:** A process that has been released from the pool of executable processes by the OS, either because it is halted or because it is aborted for some reason A process that has been released by OS either after normal termination or after abnormal termination (error).

Process Control Block (PCB)

A process control block or PCB is a data structure (a table) that holds information about a process. Every process or program that runs needs a PCB. When a user requests to run a particular program, the operating system constructs a process control block for that program. It is also known as Task Control Block (TCB). It contains many pieces of information associated with a specific process i.e. it simply serves as the respiratory for any information that may vary from process to process. The process control block typically contains:

- An ID number that identifies the process
- Pointers to the locations in the program and its data where processing last occurred
- Register contents
- States of various flags and switches
- Pointers to the upper and lower bounds of the memory required for the process
- A list of files opened by the process
- The priority of the process
- The status of all I/O devices needed by the process

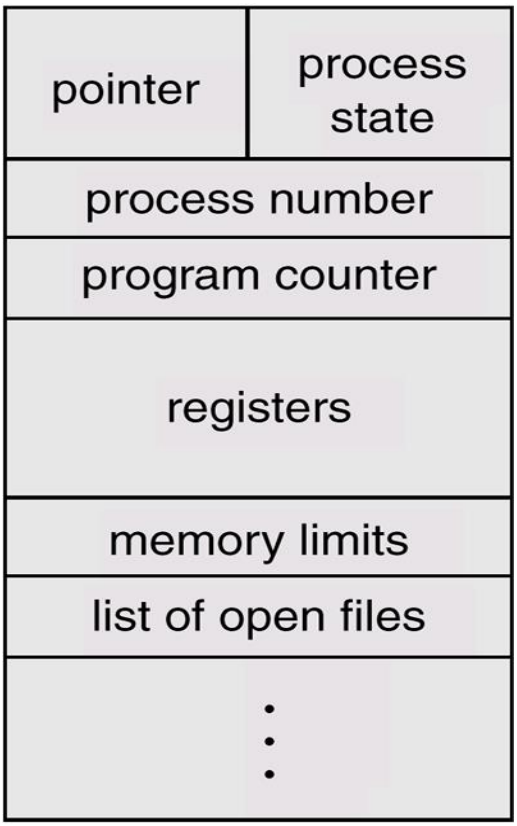


Fig Process Control Block

The information stored in the Process Control Block in given below

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

Difference Between a Program and process

Program	Process
Consists of set of instructions in programming language	It is a sequence of instruction execution
It is a static object existing in a file form	It is a dynamic object (i.e. program in execution)
Program is loaded into secondary storage device	Process is loaded into main memory
The time span is unlimited	Time span is limited
It is a passive entity	It is an active entity

2.2 Threads

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In many respect, threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel. Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or terminated). A thread consists of a program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes. Threads shares address space, program code, global variables, OS resources with other thread.

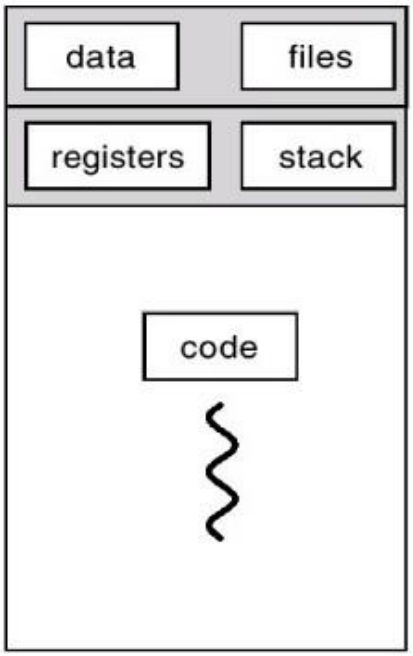


Fig Process with single thread

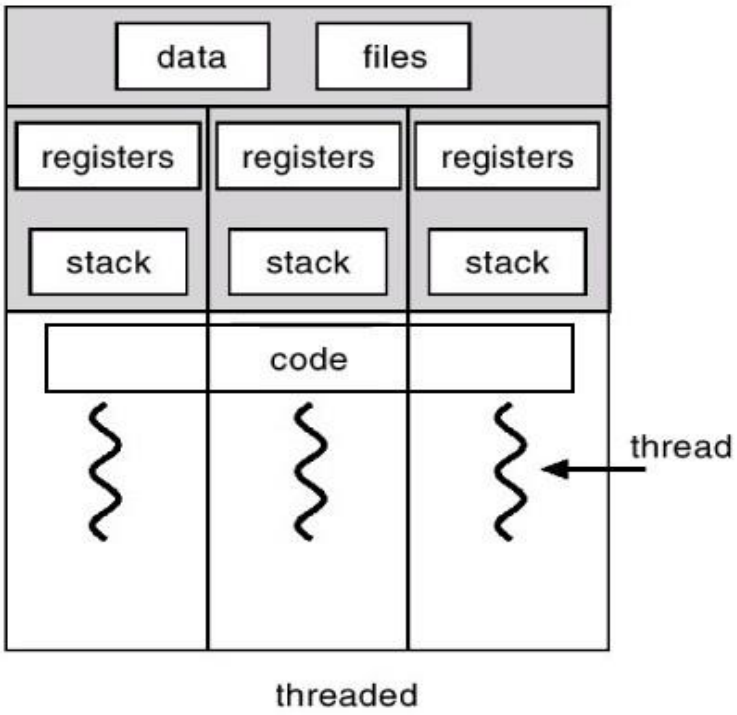


Fig: Process With multiple thread

Why Threads?

- Process with multiple threads makes a great server (e.g. print server)
- Increase responsiveness, i.e. with multiple threads in a process, if one threads blocks then other can still continue executing
- Sharing of common data reduce requirement of inter-process communication
- Proper utilization of multiprocessor by increasing concurrency
- Threads are cheap to create and use very little resources
- Context switching is fast (only have to save/reload PC, Stack, SP, Registers)

Processes Vs Threads

As we mentioned earlier that in many respect threads operate in the same way as that of processes. Some of the similarities and differences are:

Similarities

- Like processes threads share CPU and only one thread active (running) at a time.
- Like processes, threads within processes, threads within processes execute sequentially.
- Like processes, thread can create children.
- And like process, if one thread is blocked, another thread can run.

Differences:

PROCESS	THREAD
Doesn't share memory (loosely coupled)	Shares memories and files (tightly coupled)

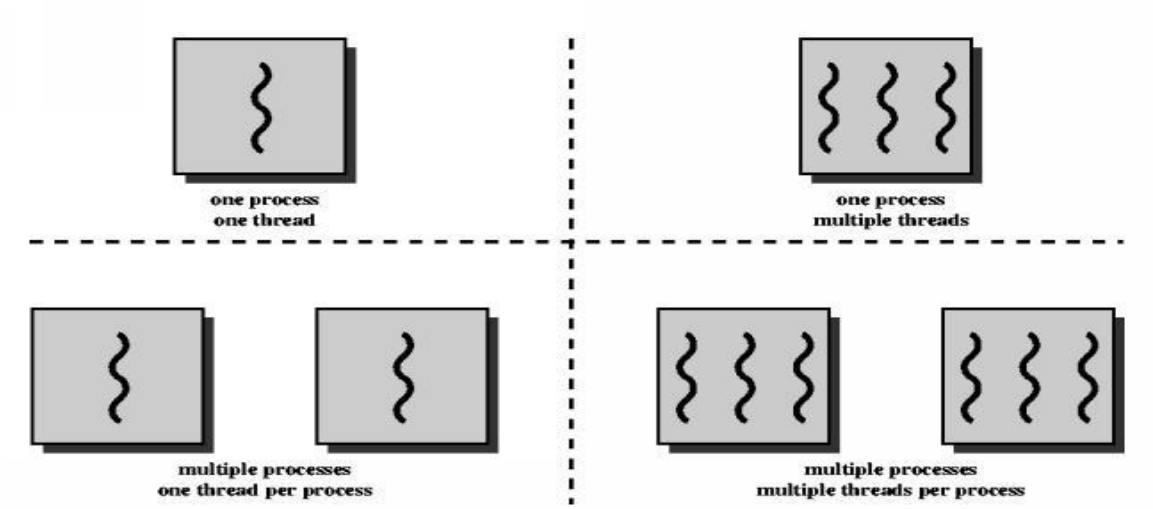
Creation is time consuming	Fast
Execution slow	Fast
More time to terminate	Less time
More time to switch between processes	Less time
System calls are required for communication	Not required
more resources are required	fewer resources are required
Not suitable for parallelism	Suitable

Multithreading

The use of multiple threads in a single program, all running at the same time and performing different tasks is known as multithreading. Multithreading is a type of execution model that allows multiple threads to exist within the context of a process such that they execute independently but share their process resources. It enables the processing of multiple threads at one time, rather than multiple processes**.**

Based on functionality, threads are put under 4 categories:

- Single Process, Single Thread (MS-DOS)
- Single Process, Multiple Threads (JAVA Runtime)
- Multiple Process, Single Thread per process (Unix)
- Multiple Process, Multiple Threads (Solaris, Unix)



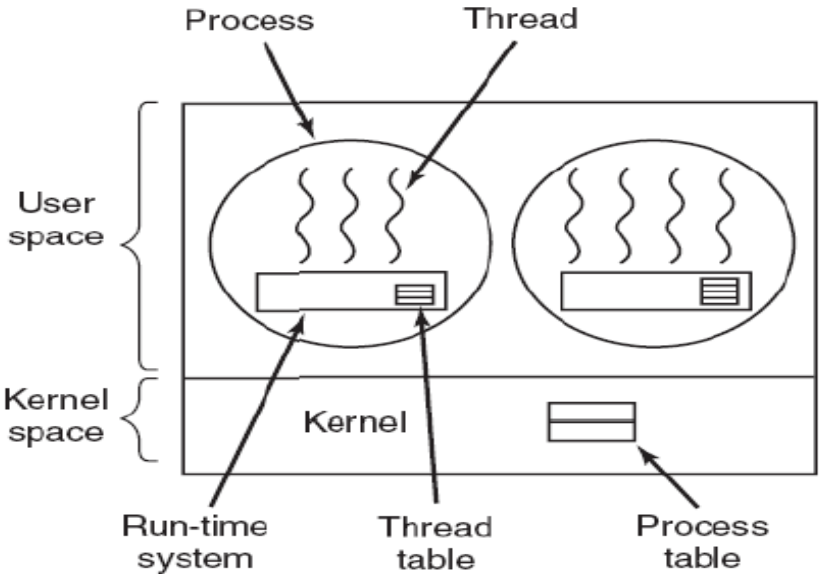
Threads Implementation

Threads are implemented in following two ways:

- User Level Threads -- User managed threads
- Kernel Level Threads -- Operating System managed threads acting on kernel, an operating system core

User Level Threads

User thread are supported at user level. In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. In fact, the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.



Advantages:

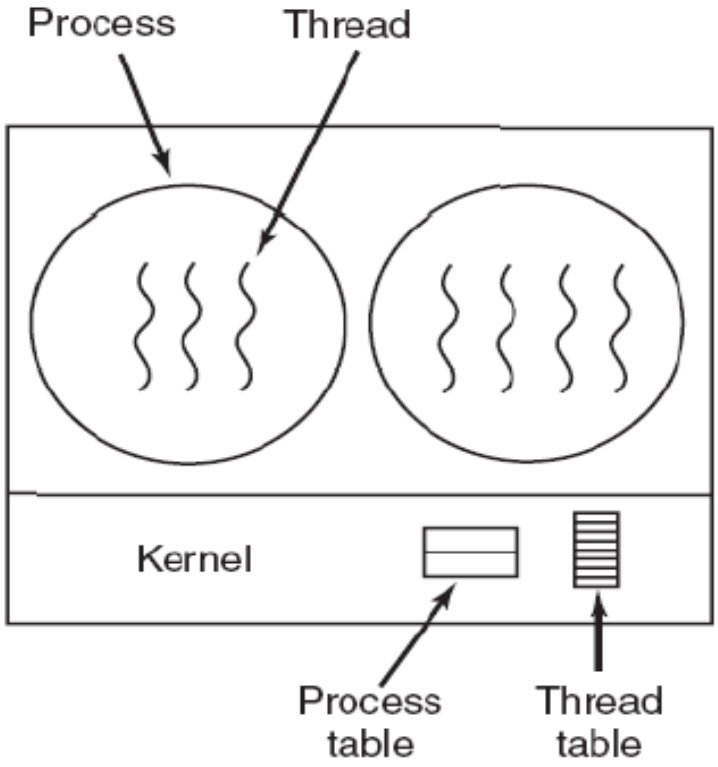
- The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads.
- User-level thread does not require modification to operating systems.
- **Simple Representation:** Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- **Simple Management:** This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- **Fast and Efficient:** Thread switching is not much more expensive than a procedure call.

Disadvantages:

- 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.
- User-level thread requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are run able threads left in the processes. For example, if one thread causes a page fault, the process blocks.

Kernel-Level Threads

In this method, the kernel knows about and manages the threads. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.



Advantages:

- 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.
- Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

- The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.
- 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.

Difference

User-Level Threads	Kernel-Level Thread
User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.

***Compiled By**: Er. Pradip Khanal*
Lecturer, ACEM

Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Combined/Hybrid Thread

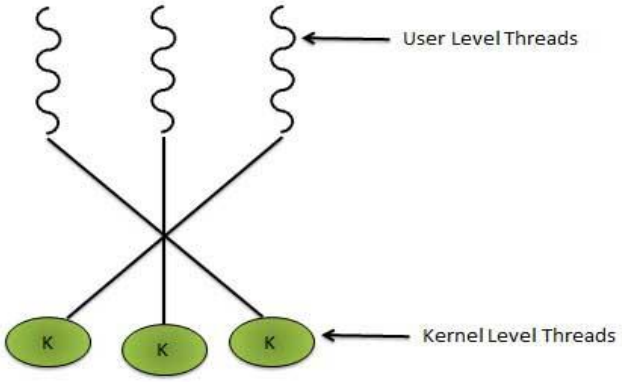
Some operating system provides a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- Many to many relationships.
- Many to one relationship.
- One to one relationship.

Many to Many Model

In this model, many user level threads multiplex to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.

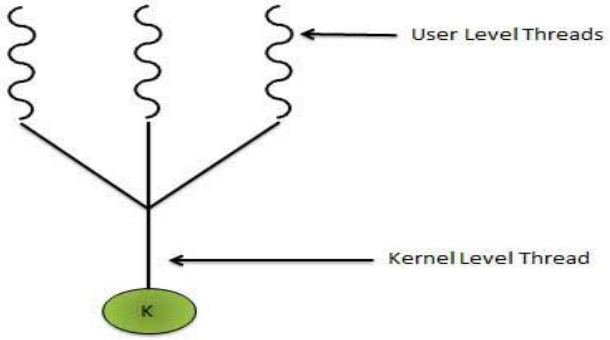
Following diagram shows the many to many models. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



Many to One Model

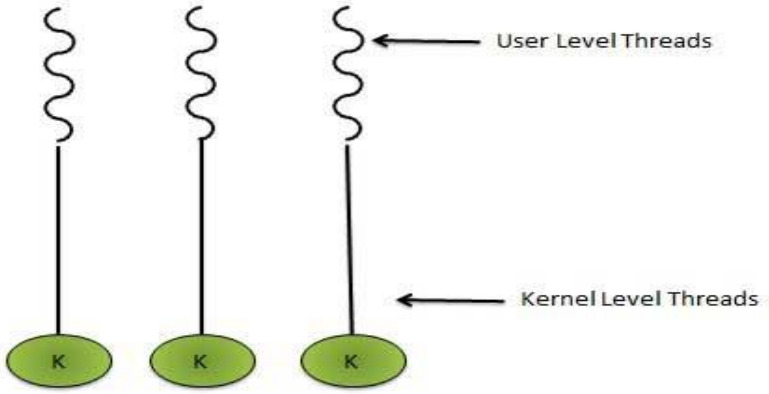
Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship modes.



One to One Model

There is one to one relationship of user level thread to the kernel level thread. This model provides more concurrency than the many to one model. It also another thread to run when a thread makes a blocking system call. It support multiple thread to execute in parallel on microprocessors. Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, Windows NT and windows 2000 use one to one relationship model.



2.4 Scheduling

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy. The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

Scheduler

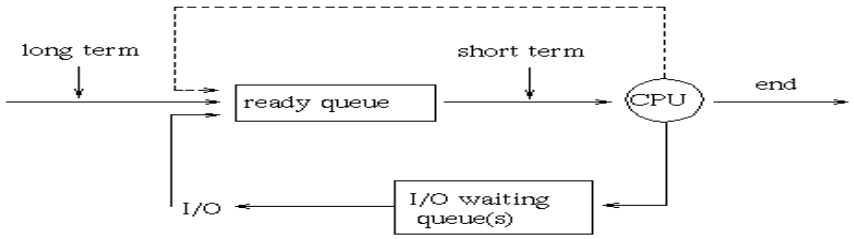
Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run.

Types of Scheduler

- Long term scheduler
- Mid - term scheduler
- Short term scheduler

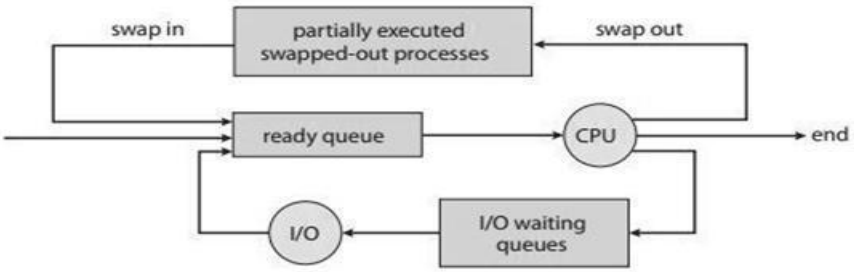
1. Long Term Scheduler

It selects the process that are to be placed in ready queue. The long term scheduler basically decides the priority in which processes must be placed in main memory. Processes of long term scheduler are placed in the ready state because in this state the process is ready to execute waiting for calls of execution from CPU which takes time that's why this is known as long term scheduler.



2. Mid – Term Scheduler

It places the blocked and suspended processes in the secondary memory of a computer system. The task of moving from main memory to secondary memory is called **swapping out**. The task of moving back a swapped out process from secondary memory to main memory is known as **swapping in**. The swapping of processes is performed to ensure the best utilization of main memory.



3. Short Term Scheduler

It decides the priority in which processes is in the ready queue are allocated the central processing unit (CPU) time for their execution. The short term scheduler is also referred as central processing unit (CPU) scheduler.

Dispatcher

The dispatcher is the module that gives control of the CPU to the process selected by the short-time scheduler (selects from among the processes that are ready to execute).
The function involves:

- Context Switching
- Switching to user mode
- Restart the execution of process

Context Switching

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features. When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

The speed of context switching depends on:

- The speed of memory
- The no. of registers that must be copied
- The existence of some special instruction

Scheduling Criteria

Scheduling criteria is also called as scheduling methodology. Key to multiprogramming is scheduling. Different CPU scheduling algorithm have different properties .The criteria used for comparing these algorithms include the following:

- **CPU Utilization:**
  Keep the CPU as busy as possible. It range from 0 to 100%. In practice, it range from 40 to 90%.

- **Throughput:**
  Throughput is the rate at which processes are completed per unit of time.

- **Turnaround time:**
  This is the how long a process takes to execute a process. It is calculated as the time gap between the submission of a process and its completion.

- **Waiting time:**
  Waiting time is the sum of the time periods spent in waiting in the ready queue.

- **Response time:**
  Response time is the time it takes to start responding from submission time. It is calculated as the amount of time it takes from when a request was submitted until the first response is produced.

- **Fairness:**
  Each process should have a fair share of CPU.

Scheduling Algorithm Goals

- **All Systems:**
  - ✓ Fairness:- Fair share of CPU to every process
  - ✓ Policy Enforcement: Stated policy is implemented properly
  - ✓ Balance: Keeping all resources equally busy

- **Batch System:**
  - ✓ Throughput:- Maximize jobs per unit time
  - ✓ Turnaround Time:- Minimization of time between submission and termination
  - ✓ CPU Utilization:- Keep CPU busy all the time

- **Interactive System:**
  - ✓ Response Time:- Quick response to request
  - ✓ Proportionality:- Meet user's expectation

- **Real Time System:**
  - ✓ Meeting Deadlines:- Avoiding losing data
  - ✓ Predictability:- Avoid quality degradation

2.4.1 Process scheduling Types

In an operating system (OS), a process scheduler performs the important activity of scheduling a process between the ready queues and waiting queue and allocating them to the CPU. The OS assigns priority to each process and maintains these queues. The scheduler selects the process from the queue and loads it into memory for execution.

There are two types of process scheduling:

- Preemptive scheduling
- Non-preemptive scheduling.

1. Preemptive Scheduling

The scheduling in which a running process can be interrupted if a high priority process enters the queue and is allocated to the CPU is called preemptive scheduling. In this case, the current process switches from the running queue to ready queue and the high priority process utilizes the CPU cycle.

2. Non-preemptive Scheduling

The scheduling in which a running process cannot be interrupted by any other process is called non-preemptive scheduling. Any other process which enters the queue has to wait until the current process finishes its CPU cycle.

Difference between Preemptive and Non Preemptive Scheduling

PREEMPTIVE SCHEDULING	NON PREEMPTIVE SCHEDULING
The resources are allocated to a process for a limited time.	Once resources are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Process can be interrupted in between.	Process cannot be interrupted till it terminates or switches to waiting state.
If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Preemptive scheduling has overheads of scheduling the processes.	Non-preemptive scheduling does not have overheads.
Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.
Preemptive scheduling is cost associated.	Non-preemptive scheduling is not cost associative.

Scheduling Algorithms

The various scheduling algorithms are as follows:

2.4.2 Scheduling in Batch System

- First-Come, First-Served (FCFS) Scheduling
- Shortest Job First (SJF) Scheduling
- Shortest Remaining Time First (SRTF) Scheduling

2.4.3 Scheduling in Interactive System

- Priority Scheduling
- Highest – Response Ratio Next (HRRN) Scheduling
- Round Robin(RR) Scheduling
- Multiple Queues
- Guaranteed Scheduling
- Lottery Scheduling
- Fair-Share Scheduling

2.4.4 Scheduling in Real Time Systems

- Real Time Scheduling

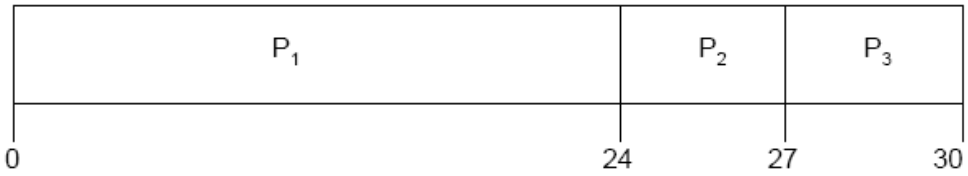
First in First Served (FCFS) Scheduling

It is simplest CPU scheduling algorithm. The FCFS scheduling algorithm is non preemptive i.e. 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.
In this technique, the process that requests the CPU first is allocated the CPU first i.e. 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 average waiting time under this technique is often quite long.

Example
Q1. Consider the following set of processes that arrives at time 0, with the length of the CPU burst given in milliseconds:

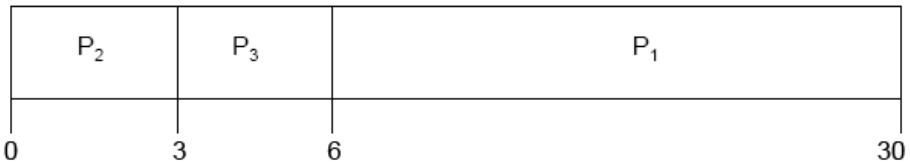
Process	Burst Time
P_1	24
P_2	3
P_3	3

(a) Suppose that the processes arrive in the order: *P1, P2,* and *P3*
The Gantt chart for the schedule is:



- ✓ Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
- ✓ Average waiting time: (0 + 24 + 27)/3 = 17 ms
- ✓ Turnaround time for *P1* = 24; *P2* = 27; *P3* = 30
- ✓ Average Turnaround time: (24 + 27 + 30)/3 = 27 ms

(b) Suppose that the processes arrive in the order: *P2, P3, and P1*
The Gantt chart for the schedule is:



- ✓ Waiting time for *P1* = 6*; P2* = 0*; P3* = 3
- ✓ Average waiting time: (6 + 0 + 3)/3 = 3 milliseconds
- ✓ Turnaround time for *P1* = 30; *P2* = 3; *P3* = 6
- ✓ Average Turnaround time: (30 + 3 + 6)/3 = 13 ms

Q2. Consider the following set of processes that arrives at time 0, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	10
P_2	1	6
P_3	3	2
P4	5	4

Find:

1. Average Turnaround Time (ATAT) (*ans* 14.25)
2. Average Waiting Time (AWT) (*ans* 8.75)
3. Waited TAT (WTAT) (*ans* 1.295)
4. Average WTAT (AWTAT) (*ans* 14.25)

Sol^n

Process	Arrival Time (T_0)	Burst Time (ΔT)	Finish Time (T_1)	TAT $=(T_1 - T_0)$	WT $=(TAT-\Delta T)$	Response Time (RT) $=(T_1-\Delta T)$
P_1	0	10	10	10	0	0

P_2	1	6	16	15	9	10
P_3	3	2	18	15	13	16
P4	5	4	22	17	13	18

1. Average Turnaround Time (ATAT) = (10+15+15+17)/4 = 14.25
2. Average Waiting Time (AWT) = (0+9+13+13) /4 = 8.75
3. Waited TAT (WTAT) $WTAT_i = TAT_i / RT_i$ where i = 1 to n
   = (10+15+15+17)/ (0+10+16+18)
   = 1.295
4. Average WTAT (AWTAT) = WTAT/n = (10+15+15+17)/4 = 14.25

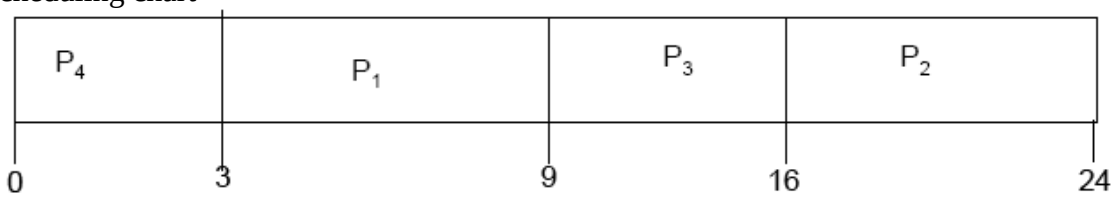
Shortest Job First (SJF) Scheduling

This technique is associated with the length of the next CPU burst of a process. When the CPU is available, it is assigned to the process that has smallest next CPU burst. If the next bursts of two processes are the same, FCFS scheduling is used.
The SJF algorithm is optimal i.e. it gives the minimum average waiting time for a given set of processes.
The real difficulty with this algorithm knows the length of next CPU request.
Let us consider following set of process with the length of the CPU burst given in milliseconds.

Examples
Q1. Consider the following set of processes that arrives at time 0, 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

SJF scheduling chart



The waiting time for process P1= 3, P2 = 16, P3 = 9 and P4 = 0 milliseconds, Thus
✓ Average waiting time = (3 + 16 + 9 + 0) / 4 = 7 milliseconds

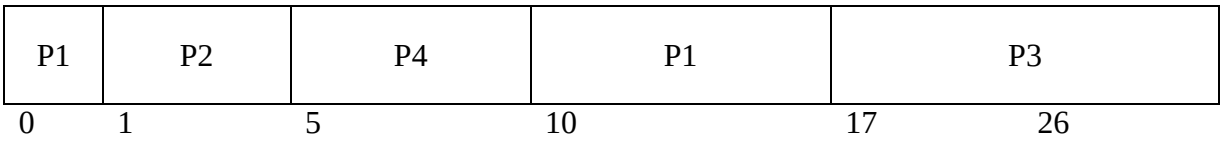
A preemptive SJF algorithm will preempt the currently executing, where as a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is also known **Shortest-Remaining-time (SRT) First Scheduling.**

***Compiled By**: Er. Pradip Khanal*
Lecturer, ACEM

Let us consider following four processes with the length of the CPU burst given in milliseconds.

Process	Arrival time	Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Gantt chart is:



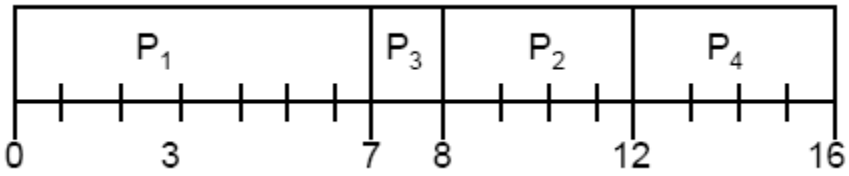
Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.

✓ The average waiting time is: = (P1 + P2 + P3 + P4) / 4
= [(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4
= (9 + 0 + 15 + 2)/4
= 26 / 4
= 6.5 milliseconds

Example: Non-Preemptive SJF

Process	Arrival time	Burst time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

• SJF (non-preemptive) Gant Chart



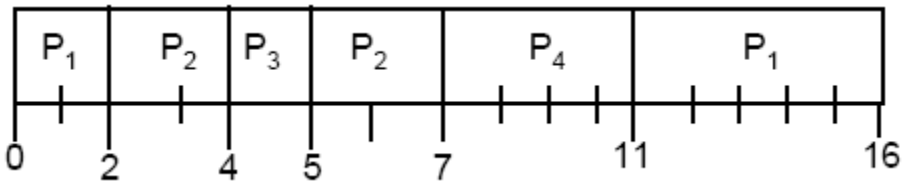
✓ Average waiting time
= (P1 + P2 + P3 + P4) / 4
= [(0 – 0) + (8 – 2) + (7 – 4) + (12 – 5)] / 4
= (0 + 6 + 3 + 7) / 4

= 4 milliseconds

Example: Preemptive SJF

Process	Arrival time	Burst time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

• SJF (preemptive) (Shortest Remaining Time First, SRTF) Gantt chart



• Average waiting time

= [(11 – 2) + (5 – 4) + (4 – 4) + (7 – 5)] / 4
= (9 + 1 + 0 +2) / 4
= 3 milliseconds

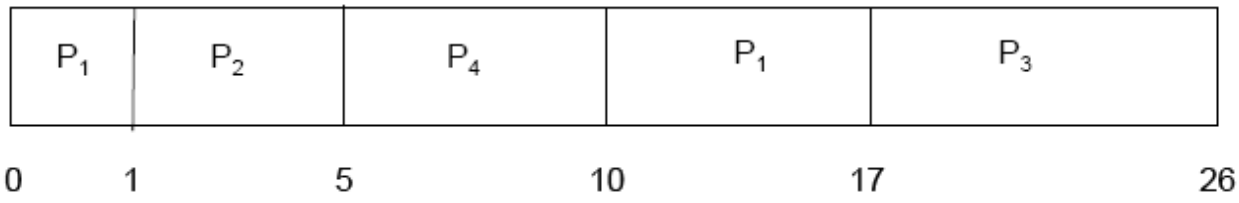
<u>Shortest Remaining Time (SRT) Scheduling</u>

It is a preemptive version of SJF algorithm where the remaining processing time is considered for assigning CPU to the next process.

Now we add the concepts of varying arrival times and preemption to the analysis

Process	Arrival time	Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

✓ *Preemptive* SJF Gantt Chart



✓ Average waiting time = [(10-1) +(1-1) +(17-2) +5-3)]/4 = 26/4 = 6.5 milliseconds

Priority Scheduling

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal priorities are scheduled in FCFS order. Priorities are generally indicated by some fixed range of numbers and there is no general method of indicating which is the highest or lowest priority, it may be either increasing or decreasing order.
Priority can be defined either internally or externally.

- ✓ Internally defined priorities use some measurable quantity 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 has been used in computing priorities.
- ✓ External priorities are set by criteria outside the OS, such as importance of process, the type and amount of funds being paid for computer user, and other 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 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.

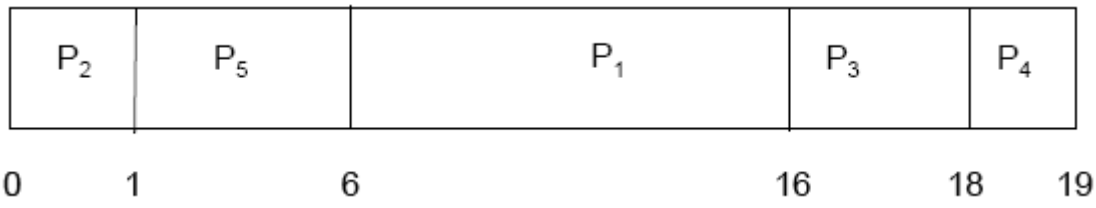
A major problem of such scheduling algorithm is indefinite blocking or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. Such scheduling can leave some low priority process waiting indefinitely. The solution for this problem is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

Q1. Consider following set of processes, assumed to have arrived at time 0 in order P1, P2, …, P5 with the length of the CPU burst given in milliseconds

Example:

Process	Burst time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Priority scheduling Gantt chart



- ✓ Average waiting time = (6 + 0 + 16 + 18 + 1) / 5
  = 41 / 5
  = 8.2 milliseconds

Round Robin Scheduling

Round Robin (RR) scheduling algorithm is designed especially for time sharing 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 **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.

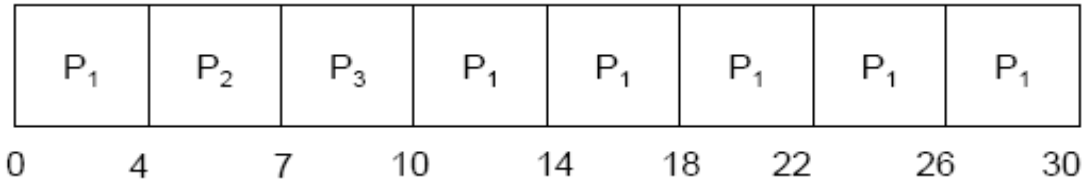
The process may have a CPU burst less than 1-time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1-time quantum, the timer will go off and will cause an interrupt to the OS. 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.

Q1. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds.
Example 1: Quantum time = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:



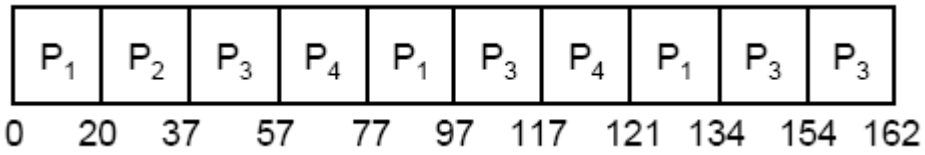
The process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first-time quantum, and the CPU is given to the next process in the queue, process P2. Process P2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then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

✓ Average time: = (P1 + P2 + P3) / 3
= [(10 – 4) + 4 + 7] / 3
= 17 / 3
= 5.66 milliseconds

Example 2: quantum = 20

Process	Burst time	Waiting time for each process
P1	53	0 + (77 – 20) + (121 – 97) = 81
P2	17	20
P3	68	37 + (97 – 57) + (134 – 117) = 94
P4	24	57 + (117 – 77) = 97

Gantt chart



✓ Average Waiting Time:
= (P1 + P2 + P3 + P4) / 4
= [{0 + (77 – 20) + (121 – 97)} + 20 + {37 + (97-57) + (134 – 117)} + {57 + (117 – 77)}] / 4
= (81+20+94+97) / 4
= 73 milliseconds

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) x q*** time units until its next time quantum. For example, there are 5 processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

Q3. Consider the following set of processes that arrive at given time, with the length of the CPU burst given in milliseconds. Calculate AWT, ATAT,

TQ= 2

Process	Arrival time	Burst time
P1	0	4
P2	1	5
P3	2	2
P4	3	1
P5	4	6
P6	6	3

Queue

P1	P2	P3	P1	P4	P5	P2	P6	P5	P2	P6	P5

Gantt Chart

P1	P2	P3	P1	P4	P5	P2	P6	P5	P2	P6	P5

0 2 4 6 8 9 11 13 15 17 18 19 21

Process	Arrival time	Burst time	FT	TAT	WT	RT
P1	0	4	8	8	4	
P2	1	5	18	17	12	
P3	2	2	6	4	2	
P4	3	1	9	6	5	
P5	4	6	21	17	11	
P6	6	3	19	13	10	

Highest-Response Ration Next (HRN) Scheduling

Highest Response Ratio Next (HRRN) scheduling is a non-preemptive discipline, in which the priority of each job is dependent on its estimated run time, and also the amount of time it has spent waiting. Jobs gain higher priority the longer they wait, which prevents indefinite postponement (process starvation).
It selects a process with the largest ratio of waiting time over service time. This guarantees that a process does not starve due to its requirements.
In fact, the jobs that have spent a long time waiting compete against those estimated to have short run times.

Priority = (waiting time + estimated runtime) / estimated runtime

(Or)

Response Ratio = (waiting time + service time) / service time

Advantages

- Improves upon SPF scheduling
- Still non-preemptive
- Considers how long process has been waiting
- Prevents indefinite postponement

Disadvantages

- Does not support external priority system. Processes are scheduled by using internal priority system.

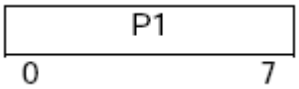
Example:

Q1. Consider the Processes with following Arrival time, Burst Time and priorities

Process	Arrival time	Burst time	Priority
P1	0	7	3 (High)
P2	2	4	1 (Low)
P3	3	4	2

Solution: HRRN

At time 0 only process p1 is available, so p1 is considered for execution



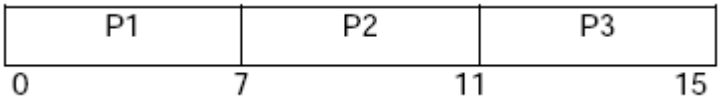
Since it is Non-preemptive, it executes process p1 completely. It takes 7 ms to complete process p1 execution.

Now, among p2 and p3 the process with highest response ratio is chosen for execution.

Response Ratio for p2 = (5 + 4) / 4 = 2.25

Response Ratio for p3 = (4 + 4) / 4 = 2

As process p2 is having highest response ratio than that of p3. Process p2 will be considered for execution and then followed by p3.



Average waiting time = 0 + (7 - 2) + (11 - 3) / 3 = 4.33

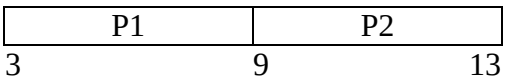
Average Turnaround time = 7 + (11 - 2) + (15 - 3) / 3 = 9.33

Q2. Consider the Processes with following Arrival time, Burst Time and priorities

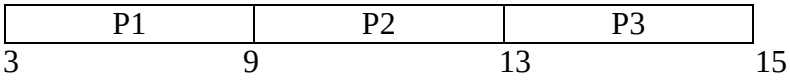
Process	Arrival time	Burst time
P1	0	3
P2	2	6
P3	4	4
P4	6	5
P5	8	2

- At time 0 only process p1 is available, so p1 is considered for execution

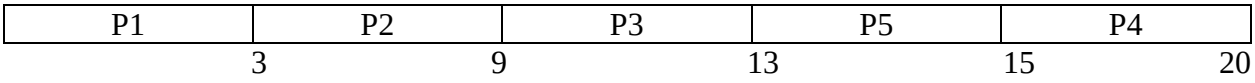
- At time 3 only process p2 is available, so p2 is considered for execution
- Since at time 9 all process has arrived so we have to calculate Response Ratio for these process



Response Ratio for P3 = (5 + 4)/4 = 2.25.
Response Ratio for P4 = (3 + 5)/5 = 1.6
Response Ratio for P5 = (1+2)/2 = 1.5



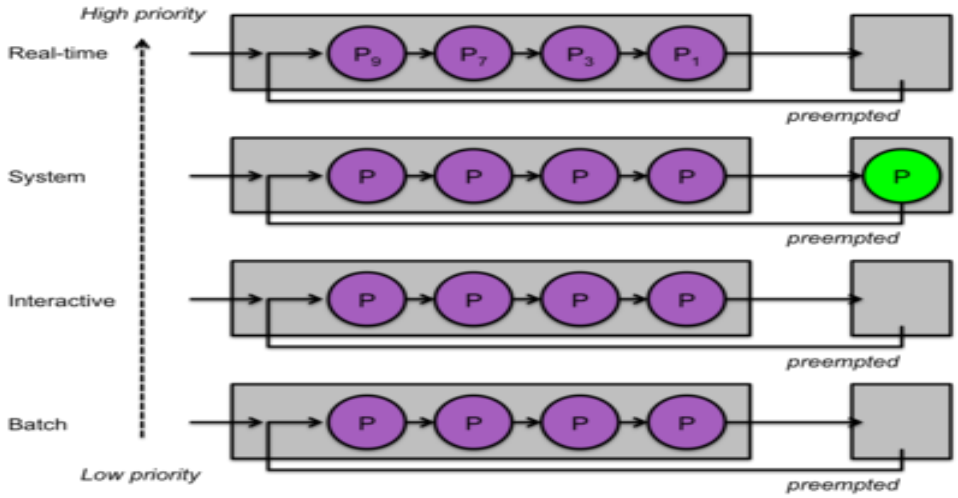
Response Ratio for P4= (7+ 5)/5= 2.4
Response Ratio for P5 = (5+2)/2 = 3.5



Multilevel Queues

Multi-level queue scheduling was created for situation in which processes are easily classified into different groups. In multilevel queue Processes are divided into different queue based on their type. Process are permanently assigned to one queue, generally based on some property of process i.e. system process, interactive, batch system, end user process, memory size, process priority and process type. Each queue has its own scheduling algorithm. For example, interactive process may use round robin scheduling method, while batch job uses the FCFS method.

In addition, there must be scheduling among the queue and is generally implemented as fixed priority preemptive scheduling. Foreground process may have higher priority over the background process.



Guaranteed Scheduling

It is a completely different approach of scheduling to make the real promises to the user. In this scheduling if there are *n* users logged on while you are working, you will receive 1/*n* of CPU time. Similarly, on the single user system with *n* process running, all things being equal, each one should get 1/*n* of CPU time.

To make good on this promise the system must keep track of how much CPU each process has had since its creation. It then computes the amount of CPU each one is entitled to, namely the time since creation divided by *n.* since the amount of CPU each time each process has actually had is also known, it is straight forward to compute the ratio of actual CPU time consumed to CPU time entitled. A ratio 0.5 means that the process had has had only half of the CPU time it should actually have had, and the ratio 2 means that the process has had twice as much as it was entitled to. The algorithm is then run to the process with the lowest ratio until its ratio has moved above its closest competitor.

Lottery Scheduling

The lottery scheduling algorithm has a very different approach to scheduling processes. It works exactly as it sounds, each process in the ready state queue is given some lottery tickets, and then the algorithm picks a winner based on a randomly generated number. The winning process is awarded a time slice, in which the process may be complete or the time slice may expire before the process completes. The winning process goes back into the ready queue, receives its tickets for the next lottery, and the cycle continues.

Although each process will receive at least one (1) ticket per lottery, processes that are expected to be shorter will receive more than one ticket, thereby increasing the chance they'll be picked. In this way, the lottery scheduler resembles SJF/SRTF, in that it gives priority to what it thinks will be shorter processes. However, it is fairer then SJF/SRTF, because it does give each process a chance (albeit a small chance) to be selected for a time slice. Starvation is therefore avoided in the lottery scheduler.

A major drawback of the lottery scheduling algorithm is that it is inherently unpredictable. Due to the random nature of the scheduler, there is no way for OS designers to know for sure what will happen in lottery scheduling, possibly producing undesirable results.

Fare-Share Scheduling

So far, we have assumed that each process is scheduled on its own, without regard to who its owner is. As a result, if user 1 starts up 9 processes and user 2 starts up 1 process, with round robin or equal priorities, user 1 will get 90% of the CPU and user 2 will get only 10% of it.

To prevent this situation, some systems take into account who owns a process before scheduling it. In this model, each user is allocated some fraction of the CPU and the scheduler picks processes in such a way as to enforce it. Thus, if two users have each been promised 50% of the CPU, they will each get that, no matter how many processes they have in existence.

As an example, consider a system with two users, each of which has been promised 50% of the CPU. User 1 has four processes, A, B, C, and D, and user 2 has only 1 process, E. If round-robin scheduling is used, a possible scheduling sequence that meets all the constraints is this one:

A E B E C E D E A E B E C E D E …

On the other hand, if user 1 is entitled to twice as much CPU time as user 2, we might get

A B E C D E A B E C D E ...

Summary

- ✓ FCFS: not fair, and average waiting time is poor;
- ✓ Round Robin: fair, but average waiting time is poor;
- ✓ SJF/SRTF: Not fair, but average waiting time is minimized assuming we can accurately predict the length of the next CPU burst. Starvation is possible;
- ✓ Multilevel Feedback Queuing: An improvement of SJF/SRTF;
- ✓ Lottery Scheduling: Fairer with low average waiting time, but less predictable.

Scheduling Algorithms

The scheduling algorithms can be divided into off-line scheduling algorithms and online scheduling algorithms.

1. **Offline**

In *offline* scheduling all decisions about scheduling is taken before the system is started and the scheduler has complete knowledge about all the tasks. During runtime the tasks are executed in a predetermined order. Offline scheduling is useful if we have a hard-real-time system with

complete know-ledge of all the tasks because then a schedule of the tasks can be made which ensures that all tasks will meet their deadlines, if such a schedule exists

2. Online

In *online* scheduling the decisions regarding how to schedule tasks are done during the runtime of the system. The scheduling decisions are based on the tasks priorities which are either assigned dynamically or statically. *Static* priority driven algorithms assign fixed priorities to the tasks before the start of the system. *Dynamic* priority driven algorithms assign the priorities to tasks during runtime. Online Scheduling types:

a) Static priority Scheduling: - Rate Monotonic Algorithm(RMA)
b) Dynamic priority Scheduling: - Earliest Deadline First Algorithm(EDA)

Rate Monotonic Algorithm(RMA)

Rate monotonic (RM) scheduling algorithm is a uniprocessor static-priority preemptive scheme. The algorithm is static-priority in the sense that all priorities are determined for all instances of tasks before runtime. What determines the priority of a task is the length of the period of the respective tasks. Tasks with short period times are assigned higher priority. RM is used to schedule periodic tasks.

The following are preconditions for the rate monotonic algorithm Periodic tasks have constant known execution timesand are ready for execution at the beginning of each period(T).

1. Deadlines(D) for tasks are at the end of each period:
$$(D = T)$$
2. The tasks are independent, that is, there is no precedence between tasks and they do not block each other.
3. Scheduling overhead due to context switches and swapping etc. are assumed to be zero.

Earliest Deadline First (EDF)

Earliest deadline first (EDF) is a dynamic priority driven scheduling algorithm which gives tasks priority based on deadline. The preconditions for RM are also valid for EDF, except the condition that deadline need to be equal to period. The task with the currently earliest deadline during runtime is assigned the highest priority. That is if a task is executing with the highest priority and another task with an earlier deadline becomes ready it receives the highest priority and therefore preempts the currently running task and begins to execute. EDF is an optimal dynamic priority driven scheduling algorithm with preemption for a real-time system on a uniprocessor. EDF is capable of achieving full processor utilization

Note: For Examples refer class note