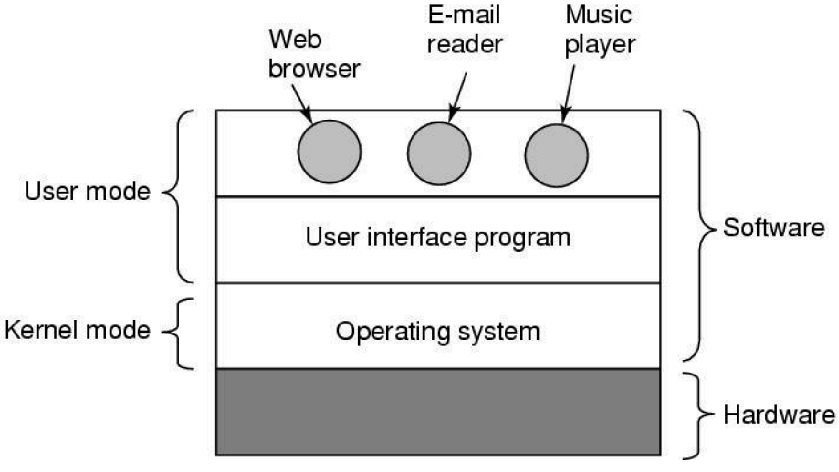
RTOS

Operating System Basics:

- The Operating System acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services
- OS manages the system resources and make them available to the user applications/tasks on a need basis.



- The primary functions of an Operating system is
  - ➢ Make the system convenient to use
  - ➢ Organize and manage the system resources efficiently and correctly

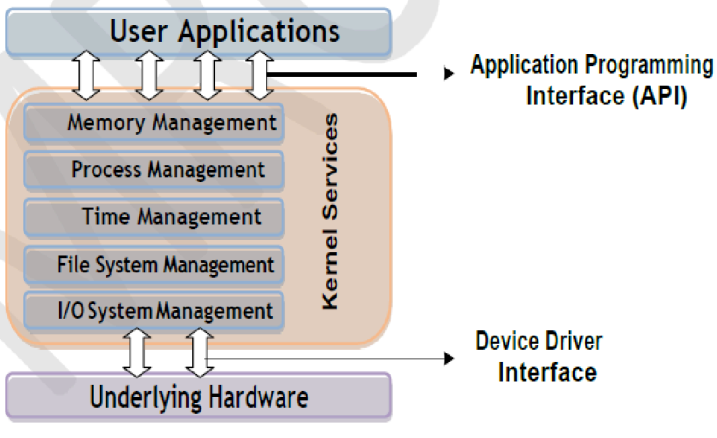


Figure 1: The Architecture of Operating System

The Kernel:

- The kernel is the core of the operating system
- It is responsible for managing the system resources and the communication among the hardware and other system services

- Kernel acts as the abstraction layer between system resources and user applications
- Kernel contains a set of system libraries and services.
- For a general purpose OS, the kernel contains different services like
  - Process Management
  - Primary Memory Management
  - File System management
  - I/O System (Device) Management
  - Secondary Storage Management
  - Protection
  - Time management
  - Interrupt Handling

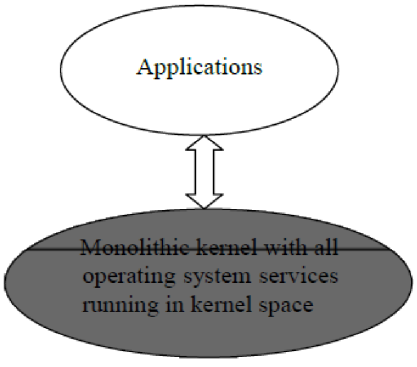
Kernel Space and User Space:

- The program code corresponding to the kernel applications/services are kept in a contiguous area (OS dependent) of primary (working) memory and is protected from the un-authorized access by user programs/applications
- The memory space at which the kernel code is located is known as '*Kernel Space'*
- All user applications are loaded to a specific area of primary memory and this memory area is referred as '*User Space*'
- The partitioning of memory into kernel and user space is purely Operating System dependent
- An operating system with virtual memory support, loads the user applications into its corresponding virtual memory space with demand paging technique
- Most of the operating systems keep the kernel application code in main memory and it is not swapped out into the secondary memory

Monolithic Kernel:

- All kernel services run in the kernel space
- All kernel modules run within the same memory space under a single kernel thread

- The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilization of the low-level features of the underlying system
- The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application

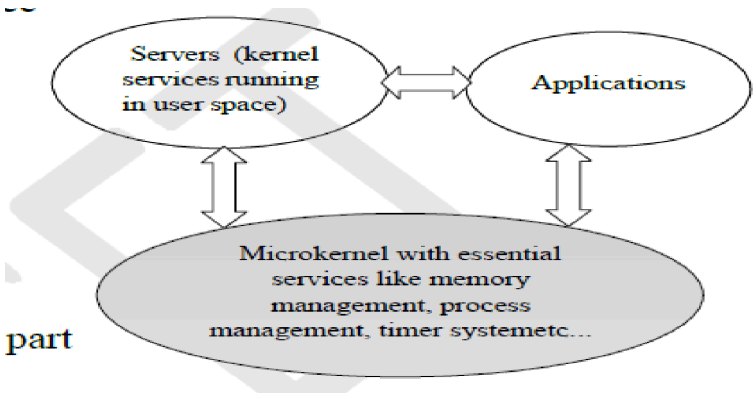


e 2: The Monolithic Kernel Model

- LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel

Microkernel

- The microkernel design incorporates only the essential set of Operating System services into the kernel
- Rest of the Operating System services are implemented in programs known as '*Servers*' which runs in user space
- The kernel design is highly modular provides OS-neutral abstraction.
- Memory management, process management, timer systems and interrupt handlers are examples of essential services, which forms the part of the microkernel
- QNX, Minix 3 kernels are examples for microkernel.



part

Benefits of Microkernel:

1. **Robustness:** If a problem is encountered in any services in server can reconfigured and re-started without the need for re-starting the entire OS.

2. **Configurability:** Any services , which run as 'server' application can be changed without need to restart the whole system.

Types of Operating Systems:

Depending on the type of kernel and kernel services, purpose and type of computing systems where the OS is deployed and the responsiveness to applications, Operating Systems are classified into

1. General Purpose Operating System (GPOS)
2. Real Time Purpose Operating System (RTOS)

General Purpose Operating System (GPOS):

- Operating Systems, **which are deployed in general computing systems**
- The kernel is more generalized and contains all the required services to execute generic applications
- **Need not be deterministic in execution behavior**
- May inject random delays into application software and thus cause slow responsiveness of an application at unexpected times
- Usually deployed in computing systems where the deterministic behavior is not an important criterion
- Personal Computer/Desktop system is a typical example for a system where GPOSs are deployed.
- Windows XP/MS-DOS etc are examples of General Purpose Operating System

2.Real Time Operating System:

- Operating Systems, which are **deployed in embedded systems demanding real-time response**
- **Deterministic in execution behavior**. Consumes only known amount of time for kernel applications
- Implements scheduling policies for executing the **highest priority task/application always**
- Implements policies and rules concerning **time-critical allocation** of a system's resources
- **Windows CE, QNX, VxWorks , MicroC/OS-II** etc are examples of Real Time Operating Systems (RTOS)

The Real Time Kernel: The kernel of a Real Time Operating System is referred as Real Time kernel. In complement to the conventional OS kernel, the Real Time kernel is highly specialized and it contains only the minimal set of services required for running the user applications/tasks. The basic functions of a Real Time kernel are

a) Task/Process management

b) Task/Process scheduling

c) Task/Process synchronization

d) Error/Exception handling

e) Memory Management

f) Interrupt handling

g) Time management

Real Time Kernel Task/Process Management:

Deals with setting up the memory space for the tasks, loading the task's code into the memory space, allocating system resources, setting up a Task Control Block (TCB) for the task and task/process termination/deletion. A Task Control Block (TCB) is used for holding the information corresponding to a task. TCB usually contains the following set of information

Task ID: **Task Identification Number**

- ***Task State:*** **The current state of the task. (E.g. State= 'Ready' for a task which is ready to execute)**
- ***Task Type*****: Task type. Indicates what is the type for this task. The task can be a hard real time or soft real time or background task.**
- ***Task Priority*****: Task priority (E.g. Task priority =1 for task with priority = 1)**
- ***Task Context Pointer*****: Context pointer. Pointer for context saving**
- ***Task Memory Pointers*****: Pointers to the code memory, data memory and stack memory for the task**
- ***Task System Resource Pointers*****: Pointers to system resources (semaphores, mutex etc) used by the task**
- ***Task Pointers:*** **Pointers to other TCBs (TCBs for preceding, next and waiting tasks)**
- ***Other Parameters*** **Other relevant task parameters**

The parameters and implementation of the TCB is kernel dependent. The TCB parameters vary across different kernels, based on the task management implementation

- **Task/Process Scheduling:** Deals with sharing the CPU among various tasks/processes. A kernel application called '*Scheduler*' handles the task scheduling. Scheduler is nothing but an

algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide a deterministic behavior.

- **Task/Process Synchronization:** Deals with synchronizing the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks.
- **Error/Exception handling:** Deals with registering and handling the errors occurred/exceptions raised during the execution of tasks. Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution etc, are examples of errors/exceptions. Errors/Exceptions can happen at the kernel level services or at task level. Deadlock is an example for kernel level exception, whereas timeout is an example for a task level exception. The OS kernel gives the information about the error in the form of a system call (API).

Memory Management:

❖ The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems

❖ The memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated memory block (initialized memory block consumes more allocation time than un- initialized memory block)

❖ Since **predictable timing** and **deterministic behavior** are the primary focus for an RTOS, RTOS achieves this by compromising the effectiveness of memory allocation

❖ RTOS generally uses '***block***' based memory allocation technique, instead of the usual dynamic memory allocation techniques used by the GPOS.

❖ RTOS kernel uses blocks of fixed size of dynamic memory and the block is allocated for a task on a need basis. The blocks are stored in a '***Free buffer Queue***'**.**

❖ Most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection to achieve predictable timing and avoid the timing overheads

❖ RTOS kernels assume that the whole design is proven correct and protection is unnecessary. Some commercial RTOS kernels allow memory protection as optional and the kernel enters a *fail-safe* mode when an illegal memory access occurs

❖ The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems

❖ A few RTOS kernels implement *Virtual Memory* concept for memory allocation if the system supports secondary memory storage (like HDD and FLASH memory).

❖ In the '*block*' based memory allocation, a **block of fixed memory** is always allocated for tasks on need basis and it is taken as a unit. Hence, there will not be any memory fragmentation issues.

❖ The memory allocation can be implemented as constant functions and thereby it consumes fixed amount of time for memory allocation. This leaves the deterministic behavior of the RTOS kernel untouched.

Interrupt Handling:

❖ Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.

❖ Interrupts can be either *Synchronous* or *Asynchronous*.

❖ Interrupts which occurs in sync with the currently executing task is known as *Synchronous* interrupts. Usually the software interrupts fall under the Synchronous Interrupt category. Divide by zero, memory segmentation error etc are examples of Synchronous interrupts.

❖ For synchronous interrupts, the interrupt handler runs in the same context of the interrupting task.

❖ Asynchronous interrupts are interrupts, which occurs at any point of execution of any task, and are not in sync with the currently executing task.

- ❖ The interrupts generated by external devices (by asserting the Interrupt line of the processor/controller to which the interrupt line of the device is connected) connected to the processor/controller, timer overflow interrupts, serial data reception/ transmission interrupts etc are examples for asynchronous interrupts.

❖ For asynchronous interrupts, the interrupt handler is usually written as separate task (Depends on OS Kernel implementation) and it runs in a different context. Hence, a context switch happens while handling the asynchronous interrupts.

❖ Priority levels can be assigned to the interrupts and each interrupts can be enabled or disabled individually.

❖ Most of the RTOS kernel implements '*Nested Interrupts*' architecture. Interrupt nesting allows the pre-emption (interruption) of an Interrupt Service Routine (ISR), servicing an interrupt, by a higher priority interrupt.

Time Management:

❖ Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.

❖ Accurate time management is essential for providing precise time reference for all applications

❖ The time reference to kernel is provided by a high-resolution Real Time Clock (RTC) hardware chip (hardware timer)

❖ The hardware timer is programmed to interrupt the processor/controller at a fixed rate. This timer interrupt is referred as 'Timer tick'

❖ The 'Timer tick' is taken as the timing reference by the kernel. The 'Timer tick' interval may vary depending on the hardware timer. Usually the 'Timer tick' varies in the microseconds range

❖ The time parameters for tasks are expressed as the multiples of the 'Timer tick'

❖ The System time is updated based on the 'Timer tick'

- ❖ If the System time register is 32 bits wide and the 'Timer tick' interval is 1microsecond, the System time register will reset in

$32 * 10^6 / (24 * 60 * 60) = 49700 \approx 0.0497$ Days = 1.19 Hours

If the 'Timer tick' interval is 1 millisecond, the System time register will reset in

$32 * 10^3 / (24 * 60 * 60) = 497 = 49.7$ Days ≈ 50 Days

- The '*Timer tick*' interrupt is handled by the 'Timer Interrupt' handler of kernel. The '*Timer tick*' interrupt can be utilized for implementing the following actions.
- Save the current context (Context of the currently executing task)
- Increment the System time register by one. Generate timing error and reset the System time register if the timer tick count is greater than the maximum range available for System time register
- Update the timers implemented in kernel (Increment or decrement the timer registers for each timer depending on the count direction setting for each register. Increment registers with count direction setting = '*count up*' and decrement registers with count direction setting = '*count down*')
- Activate the periodic tasks, which are in the idle state
- Invoke the scheduler and schedule the tasks again based on the scheduling algorithm
- Delete all the terminated tasks and their associated data structures (TCBs)
- Load the context for the first task in the ready queue. Due to the re- scheduling, the ready task might be changed to a new one from the task, which was pre-empted by the 'Timer Interrupt' task

Hard Real-time System:

- A Real Time Operating Systems which strictly adheres to the timing constraints for a task
- A Hard Real Time system must meet the deadlines for a task without any slippage
- Missing any deadline may produce catastrophic results for Hard Real Time Systems, including permanent data lose and irrecoverable damages to the system/users
- Emphasize on the principle '*A late answer is a wrong answer*'
- Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples of Hard Real Time Systems
- As a rule of thumb, Hard Real Time Systems does not implement the virtual memory model for handling the memory. This eliminates the delay in swapping in and out the code corresponding to the task to and from the primary memory

- The presence of *Human in the loop* (*HITL*) for tasks introduces un- expected delays in the task execution. Most of the Hard Real Time Systems are automatic and does not contain a 'human in the loop'

Soft Real-time System:

- Real Time Operating Systems that does not guarantee meeting deadlines, but, offer the best effort to meet the deadline
- Missing deadlines for tasks are acceptable if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS)
- A Soft Real Time system emphasizes on the principle '*A late answer is an acceptable answer, but it could have done bit faster'*
- Soft Real Time systems most often have a '*human in the loop (HITL)*'
  - Automatic Teller Machine (ATM) is a typical example of Soft Real Time System. If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens.
  - An audio video play back system is another example of Soft Real Time system. No potential damage arises if a sample comes late by fraction of a second, for play back.

Tasks, Processes & Threads :

- In the Operating System context, a task is defined as the program in execution and the related information maintained by the Operating system for the program
- Task is also known as '*Job*' in the operating system context
- A program or part of it in execution is also called a '*Process*'
- The terms '*Task*', '*job*' and '*Process*' refer to the same entity in the Operating System context and most often they are used interchangeably
- A process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange etc

The structure of a Processes

- The concept of '*Process*' leads to concurrent execution (pseudo parallelism) of tasks and thereby the efficient utilization of the CPU and other system resources
- Concurrent execution is achieved through the sharing of CPU among the processes.
- A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process

- A process, which inherits all the properties of the CPU, can be considered as a virtual processor, awaiting its turn to have its properties switched into the physical processor

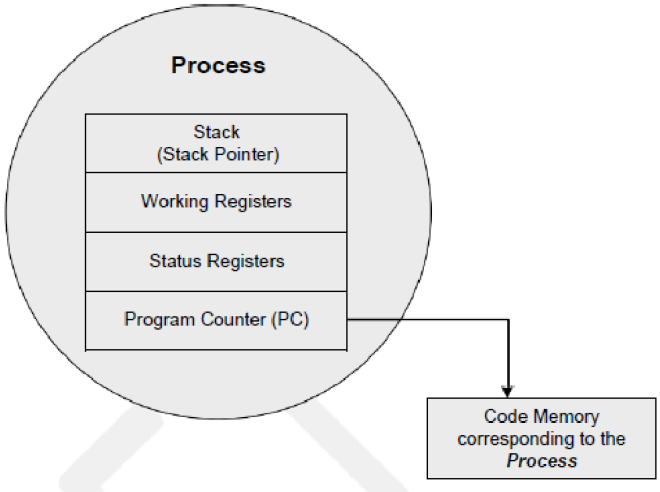
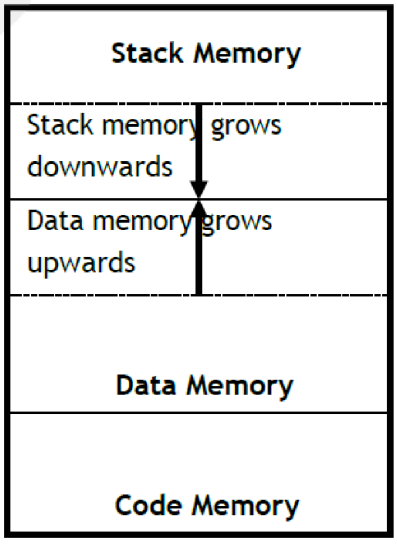


Figure: 4 Structure of a Process

- When the process gets its turn, its registers and Program counter register becomes mapped to the physical registers of the CPU

Memory organization of Processes:

- The memory occupied by the *process* is segregated into three regions namely; Stack memory, Data memory and Code memory
- The 'Stack' memory holds all temporary data such as variables local to the process
- Data memory holds all global data for the process
- The code memory contains the program code (instructions) corresponding to the process
- On loading a process into the main memory, a specific area of memory is allocated for the process
- The stack memory usually starts at the highest memory address from the memory area allocated for the process (Depending on the OS kernel implementation)

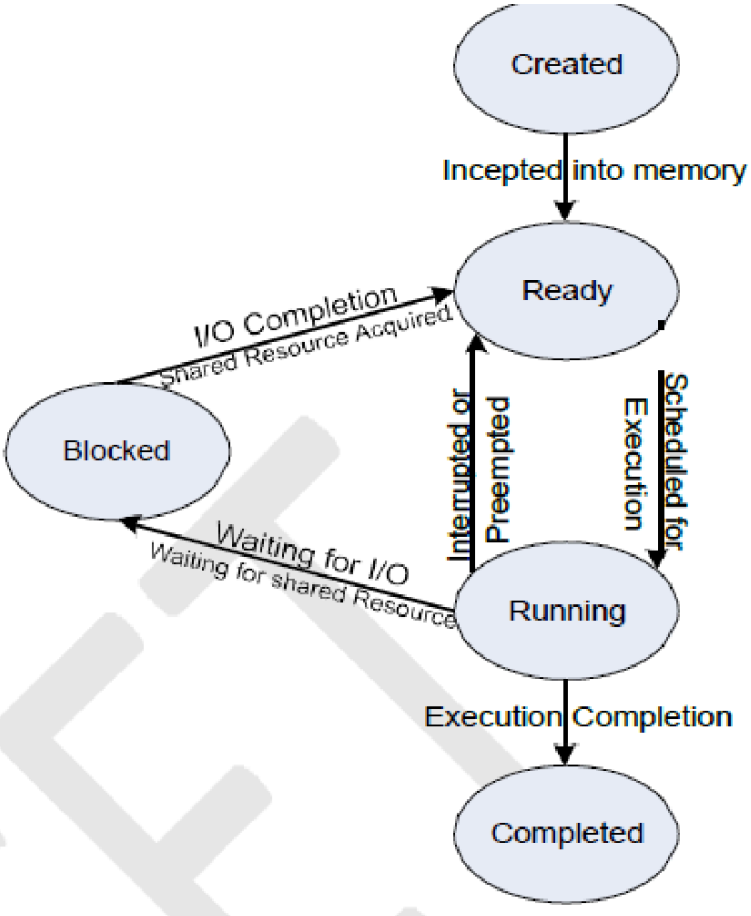


Process States & State Transition

- The creation of a process to its termination is not a single step operation
- The process traverses through a series of states during its transition from the newly created state to the terminated state
- The cycle through which a process changes its state from '*newly created*' to '*execution completed*' is known as '*Process Life Cycle*'. The various states through which a process traverses through during a Process Life Cycle indicates the current status of the process with respect to time and also provides information on what it is allowed to do next

Process States & State Transition:

- **Created State:** The state at which a process is being created is referred as 'Created State'. The Operating System recognizes a process in the '*Created State*' but no resources are allocated to the process
- **Ready State:** The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as '*Ready State*'. At this stage, the process is placed in the '*Ready list*' queue maintained by the OS
- **Running State:** The state where in the source code instructions corresponding to the process is being executed is called '*Running State*'. Running state is the state at which the process execution happens
- **Blocked State/Wait State:** Refers to a state where a running process is temporarily suspended from execution and does not have immediate access to resources. The blocked state might have invoked by various conditions like- the process enters a wait state for an event to occur (E.g. Waiting for user inputs such as keyboard input) or waiting for getting access to a shared resource like semaphore, mutex etc
- **Completed State:** A state where the process completes its execution



❖ **Figure 6.Process states and State transition**

■ The transition of a process from one state to another is known as '*State transition*'

■ When a process changes its state from Ready to running or from running to blocked or terminated or from blocked to running, the CPU allocation for the process may also change

Threads

- ❖ A *thread* is the primitive that can execute code
- ❖ A *thread* is a single sequential flow of control within a process
- ❖ '*Thread*' is also known as lightweight process
- ❖ A process can have many threads of execution
- ❖ Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area
- ❖ Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack

The Concept of multithreading

Use of multiple threads to execute a process brings the following advantages.

• Better memory utilization.
• Since the process is split into different threads, when one thread enters a wait state, the CPU can be utilized by other.

- Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.

- threads of the process that do not require the event, which the other thread is waiting, for processing. This speeds up the execution of the process.

- Efficient CPU utilization. The CPU is engaged all time.

Thread	Process
Thread is a single unit of execution and is part of process.	Process is a program in execution and contains one or more threads.
A thread does not have its own data memory and heap memory. It shares the data memory and heap memory with other threads of the same process.	Process has its own code memory, data memory and stack memory.
A thread cannot live independently; it lives within the process.	A process contains at least one thread.
There can be multiple threads in a process. The first thread (main thread) calls the main function and occupies the start of the stack memory of the process.	Threads within a process share the code, data and heap memory. Each thread holds separate memory area for stack (shares the total stack memory of the process).
Threads are very inexpensive to create	Processes are very expensive to create. Involves many OS overhead.
Context switching is inexpensive and fast	Context switching is complex and involves lot of OS overhead and is comparatively slower.
If a thread expires, its stack is reclaimed by the process.	If a process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also dies.

Advantages of Threads:

1. **Better memory utilization:** Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.

2. **Efficient CPU utilization:** The CPU is engaged all time.

3. Speeds up the execution of the process: The process is split into different threads, when one thread enters a wait state, the CPU can be utilized by other threads of the process that do not require the event, which the other thread is waiting, for processing.

Multiprocessing & Multitasking

- The ability to execute multiple processes simultaneously is referred as *multiprocessing*
- Systems which are capable of performing multiprocessing are known as *multiprocessor* systems
- *Multiprocessor* systems possess multiple CPUs and can execute multiple processes simultaneously

- The ability of the Operating System to have multiple programs in memory, which are ready for execution, is referred as *multiprogramming*

- *Multitasking* refers to the ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process

- *Multitasking* involves '*Context switching*', '*Context saving*' and '*Context retrieval*'

- *Context switching* refers to the switching of execution context from task to other

- When a task/process switching happens, the current context of execution should be saved to (*Context saving*) retrieve it at a later point of time when the CPU executes the process, which is interrupted currently due to execution switching

- During context switching, the context of the task to be executed is retrieved from the saved context list. This is known as *Context retrieval*

Multitasking – Context Switching:

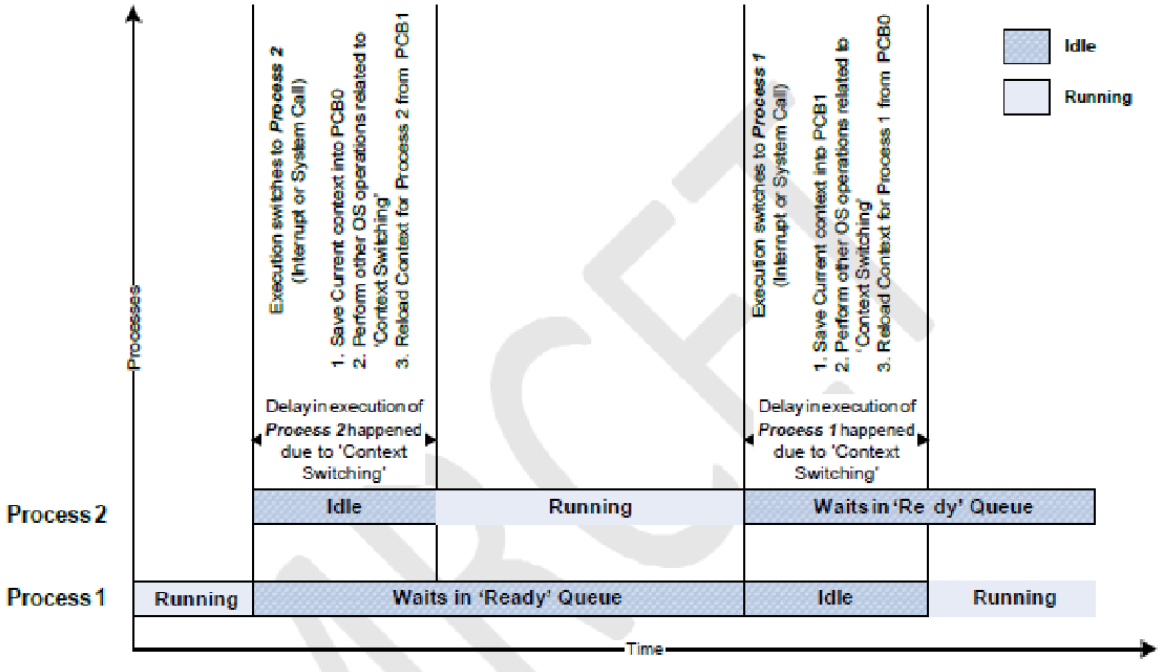


Figure 9 Context Switching

Multiprogramming: The ability of the Operating System to have multiple programs in memory, which are ready for execution, is referred as multiprogramming.

Types of Multitasking :

Depending on how the task/process execution switching act is implemented, multitasking can is classified into

Co-operative Multitasking: Co-operative multitasking is the most primitive form of multitasking in which a task/process gets a chance to execute only when the currently executing task/process voluntarily relinquishes the CPU. In this method, any task/process can avail the CPU as much time as it wants. Since this type of implementation involves the mercy of the tasks each other for getting the CPU time for execution, it is known as co-operative multitasking. If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU

• **Preemptive Multitasking:** Preemptive multitasking ensures that every task/process gets a chance to execute. When and how much time a process gets is dependent on the implementation of the preemptive scheduling. As the name indicates, in preemptive

multitasking, the currently running task/process is preempted to give a chance to other tasks/process to execute. The preemption of task may be based on time slots or task/process priority

- **Non-preemptive Multitasking:** The process/task, which is currently given the CPU time, is allowed to execute until it terminates (enters the 'Completed' state) or enters the 'Blocked/Wait' state, waiting for an I/O. The co- operative and non-preemptive multitasking differs in their behavior when they are in the 'Blocked/Wait' state. In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the 'Blocked/Wait' sate, waiting for an I/O, or a shared resource access or an event to occur, whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O.

Task Scheduling:

- In a multitasking system, there should be some mechanism in place to share the CPU among the different tasks and to decide which process/task is to be executed at a given point of time

- Determining which task/process is to be executed at a given point of time is known as task/process scheduling

- Task scheduling forms the basis of multitasking

- Scheduling policies forms the guidelines for determining which task is to be executed when

- The scheduling policies are implemented in an algorithm and it is run by the kernel as a service

- The kernel service/application, which implements the scheduling algorithm, is known as '*Scheduler*'

- The task scheduling policy can be *pre-emptive*, *non-preemptive* or *co- operative*

- Depending on the scheduling policy the process scheduling decision may take place when a process switches its state to

  ➢ '*Ready*' state from '*Running*' state

➢ '*Blocked/Wait*' state from '*Running*' state

➢ '*Ready*' state from '*Blocked/Wait*' state

➢ '*Completed*' state

Task Scheduling - Scheduler Selection:

The selection of a scheduling criteria/algorithm should consider

CPU Utilization: The scheduling algorithm should always make the CPU utilization high. CPU utilization is a direct measure of how much percentage of the CPU is being utilized.

Throughput: This gives an indication of the number of processes executed per unit of time. The throughput for a good scheduler should always be higher.

Turnaround Time: It is the amount of time taken by a process for completing its execution. It includes the time spent by the process for waiting for the main memory, time spent in the ready queue, time spent on completing the I/O operations, and the time spent in execution. The turnaround time should be a minimum for a good scheduling algorithm.

Waiting Time: It is the amount of time spent by a process in the '*Ready*' queue waiting to get the CPU time for execution. The waiting time should be minimal for a good scheduling algorithm.

Response Time: It is the time elapsed between the submission of a process and the first response. For a good scheduling algorithm, the response time should be as least as possible

Task Scheduling - Queues

The various queues maintained by OS in association with CPU scheduling are

• Job Queue: Job queue contains all the processes in the system

• Ready Queue: Contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution. The Ready queue is empty when there is no process ready for running.

• Device Queue: Contains the set of processes, which are waiting for an I/O device

Task Scheduling – Task transition through various Queues

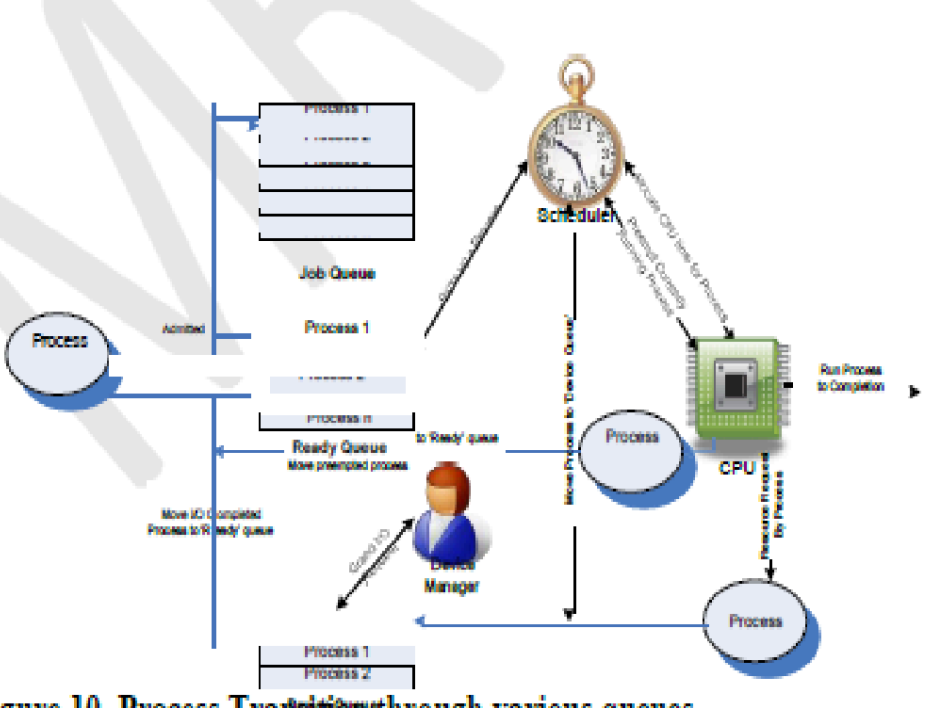


Figure 10. Process Transition through various queues

Non-preemptive scheduling – First Come First Served (FCFS)/First In First Out (FIFO) Scheduling:

- Allocates CPU time to the processes based on the order in which they enters the '*Ready*' queue
- The first entered process is serviced first
- It is same as any real world application where queue systems are used; E.g. Ticketing

Drawbacks:

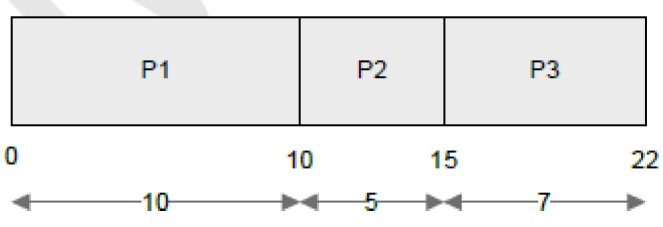
➢ Favors monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task.

➢ In general, FCFS favors CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilization.

➢ The average waiting time is not minimal for FCFS scheduling algorithm

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes).

Solution: The sequence of execution of the processes by the CPU is represented as



Assuming the CPU is readily available at the time of arrival of P1, P1 starts executing without any waiting in the 'Ready' queue. Hence the waiting time for P1 is zero.

Waiting Time for P1 = 0 ms (P1 starts executing first)

Waiting Time for P2 = 10 ms (P2 starts executing after completing P1)

Waiting Time for P3 = 15 ms (P3 starts executing after completing P1 and P2)

Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P2+P3)) / 3

= (0+10+15)/3 = 25/3 = 8.33 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 15 ms (-Do-)
Turn Around Time (TAT) for P3 = 22 ms (-Do-)
Average Turn Around Time= (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P1+P2+P3)) / 3

= (10+15+22)/3 = 47/3
= 15.66 milliseconds

Non-preemptive scheduling – Last Come First Served (LCFS)/Last In First Out (LIFO) Scheduling:

- ❖ Allocates CPU time to the processes based on the order in which they are entered in the '*Ready*' queue
- ❖ The last entered process is serviced first

- ❖ LCFS scheduling is also known as Last In First Out (LIFO) where the process, which is put last into the '*Ready*' queue, is serviced first

Drawbacks:

➢ Favors monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task

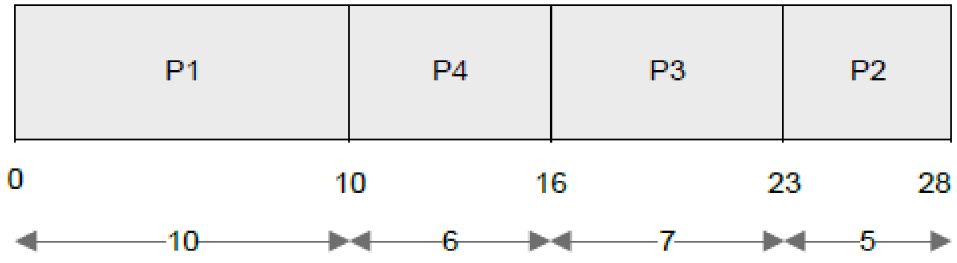
➢ In general, LCFS favors CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilization.

➢ The average waiting time is not minimal for LCFS scheduling algorithm

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3 (Assume only P1 is present in the '*Ready*' queue when the scheduler picks up it and P2, P3 entered '*Ready*' queue after that). Now a new process P4 with estimated completion time 6ms enters the 'Ready' queue after 5ms of scheduling P1. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes).Assume all the processes contain only CPU operation and no I/O operations are involved.

Solution: Initially there is only P1 available in the Ready queue and the scheduling sequence will be P1, P3, P2. P4 enters the queue during the execution of P1 and becomes the last

process entered the '*Ready*' queue. Now the order of execution changes to P1, P4, P3, and P2 as given below.



The waiting time for all the processes are given as
Waiting Time for P1 = 0 ms (P1 starts executing first)
Waiting Time for P4 = 5 ms (P4 starts executing after completing P1. But P4 arrived after 5ms of execution of P1.
Hence its waiting time = Execution start time – Arrival Time = 10-5 = 5ms)

Waiting Time for P3 = 16 ms (P3 starts executing after completing P1 and P4)
Waiting Time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P4+P3+P2)) / 4
= (0 + 5 + 16 + 23)/4 = 44/4
= 11 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 11 ms (Time spent in Ready Queue +
Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time
= (10-5) + 6 = 5 + 6
Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P1+P4+P3+P2)) / 4
= (10+11+23+28)/4 = 72/4
= 18 milliseconds

Non-preemptive scheduling – Shortest Job First (SJF) Scheduling.

- Allocates CPU time to the processes based on the execution completion time for tasks
- The average waiting time for a given set of processes is minimal in SJF scheduling
- Optimal compared to other non-preemptive scheduling like FCFS

Drawbacks:

➢ A process whose estimated execution completion time is high may not get a chance to execute if more and more processes with least estimated execution time enters the '*Ready*' queue before the process with longest estimated execution time starts its execution

➢ May lead to the 'Starvation' of processes with high estimated completion time

➢ Difficult to know in advance the next shortest process in the '*Ready*' queue for scheduling since new processes with different estimated execution time keep entering the '*Ready*' queue at any point of time.

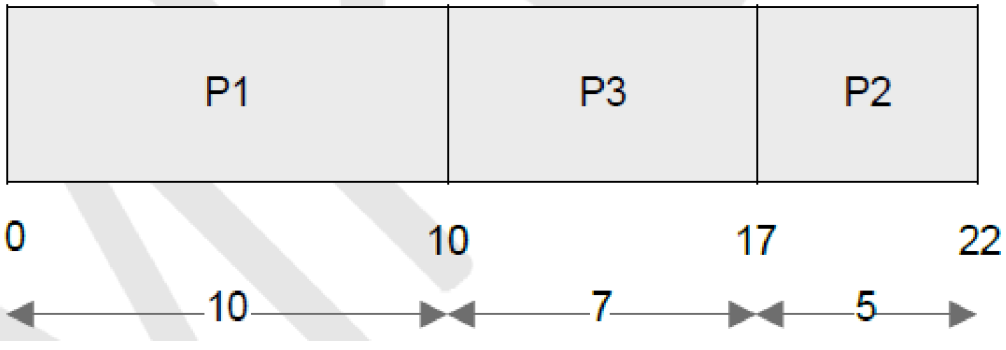
Non-preemptive scheduling – Priority based Scheduling

- ❖ A priority, which is unique or same is associated with each task
- ❖ The priority of a task is expressed in different ways, like a priority number, the time required to complete the execution etc.
- ❖ In number based priority assignment the priority is a number ranging from 0 to the maximum priority supported by the OS. The maximum level of priority is OS dependent.
- ❖ Windows CE supports 256 levels of priority (0 to 255 priority numbers, with 0 being the highest priority)
- ❖ The priority is assigned to the task on creating it. It can also be changed dynamically (If the Operating System supports this feature)
- ❖ The non-preemptive priority based scheduler sorts the 'Ready' queue based on the priority and picks the process with the highest level of priority for execution

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 0, 3, 2 (0- highest priority, 3 lowest priority) respectively

enters the ready queue together. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in priority based scheduling algorithm.

Solution: The scheduler sorts the '*Ready*' queue based on the priority and schedules the process with the highest priority (P1 with priority number 0) first and the next high priority process (P3 with priority number 2) as second and so on. The order in which the processes are scheduled for execution is represented as



The waiting time for all the processes are given as
Waiting Time for P1 = 0 ms (P1 starts executing first)
Waiting Time for P3 = 10 ms (P3 starts executing after completing P1)
Waiting Time for P2 = 17 ms (P2 starts executing after completing P1 and P3)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P3+P2)) / 3
= (0+10+17)/3 = 27/3
= 9 milliseconds
Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P3 = 17 ms (-Do-)
Turn Around Time (TAT) for P2 = 22 ms (-Do-)
Average Turn Around Time= (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P1+P3+P2)) / 3
= (10+17+22)/3 = 49/3
= 16.33 milliseconds

Drawbacks:

➢ Similar to SJF scheduling algorithm, non-preemptive priority based algorithm also possess the drawback of '*Starvation*' where a process whose priority is low may not get a chance to execute if more and more processes with higher priorities enter the '*Ready*' queue before the process with lower priority starts its execution.

➢ '*Starvation*' can be effectively tackled in priority based non-preemptive scheduling by dynamically raising the priority of the low priority task/process which is under starvation (waiting in the ready queue for a longer time for getting the CPU time) .The technique of gradually raising the priority of processes which are waiting in the 'Ready' queue as time progresses, for preventing '*Starvation*', is known as '*Aging*'.

Preemptive scheduling:

❖ Employed in systems, which implements preemptive multitasking model

❖ Every task in the '*Ready*' queue gets a chance to execute. When and how often each process gets a chance to execute (gets the CPU time) is dependent on the type of preemptive scheduling algorithm used for scheduling the processes

❖ The scheduler can preempt (stop temporarily) the currently executing task/process and select another task from the '*Ready*' queue for execution

❖ When to pre-empt a task and which task is to be picked up from the '*Ready*' queue for execution after preempting the current task is purely dependent on the scheduling algorithm

❖ A task which is preempted by the scheduler is moved to the '*Ready*' queue. The act of moving a '*Running*' process/task into the '*Ready*' queue by the scheduler, without the processes requesting for it is known as '*Preemption*'

❖ Time-based preemption and priority-based preemption are the two important approaches adopted in preemptive scheduling

Preemptive scheduling – Preemptive SJF Scheduling/ Shortest Remaining Time (SRT):

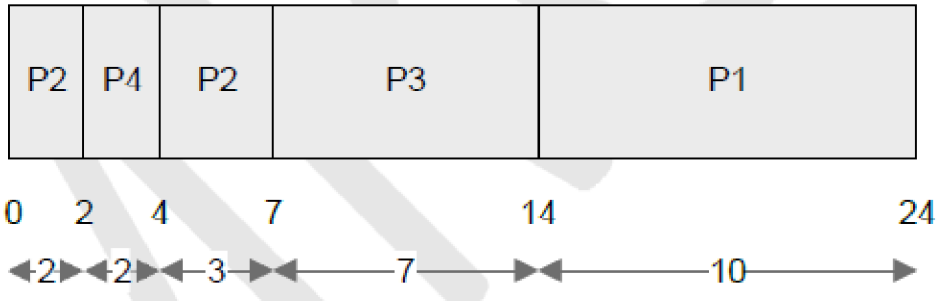
The *non preemptive SJF* scheduling algorithm sorts the 'Ready' queue only after the current process completes execution or enters wait state, whereas the *preemptive SJF* scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated execution time of the currently executing process

• If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for execution
Always compares the execution completion time (ie the remaining execution time for the new process) of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process and schedules the process with shortest remaining time for execution.

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. A new process P4 with

estimated completion time 2ms enters the 'Ready' queue after 2ms. Assume all the processes contain only CPU operation and no I/O operations are involved.

Solution: At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the SRT scheduler picks up the process with the Shortest remaining time for execution completion (In this example P2 with remaining time 5ms) for scheduling. Now process P4 with estimated execution completion time 2ms enters the '*Ready*' queue after 2ms of start of execution of P2. The processes are re-scheduled for execution in the following order

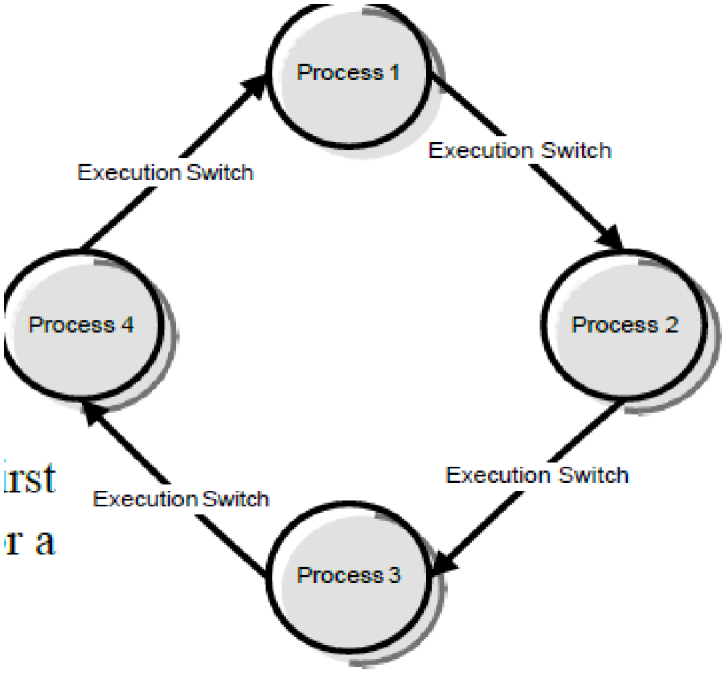


The waiting time for all the processes are given as
Waiting Time for P2 = 0 ms + (4 -2) ms = 2ms (P2 starts executing first and is interrupted by P4 and has to wait till the completion of P4 to get the next CPU slot)

Waiting Time for P4 = 0 ms (P4 starts executing by preempting P2 since the execution time for completion of P4 (2ms) is less than that of the Remaining time for execution completion of P2 (Here it is 3ms))
Waiting Time for P3 = 7 ms (P3 starts executing after completing P4 and P2)
Waiting Time for P1 = 14 ms (P1 starts executing after completing P4, P2 and P3)
Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P4+P2+P3+P1)) / 4
= (0 + 2 + 7 + 14)/4 = 23/4
= 5.75 milliseconds
Turn Around Time (TAT) for P2 = 7 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 2 ms
(Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (2-2) + 2)
Turn Around Time (TAT) for P3 = 14 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P1 = 24 ms (Time spent in Ready Queue + Execution Time)
Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P2+P4+P3+P1)) / 4
= (7+2+14+24)/4 = 47/4
= 11.75 milliseconds

Preemptive scheduling – Round Robin (RR) Scheduling:

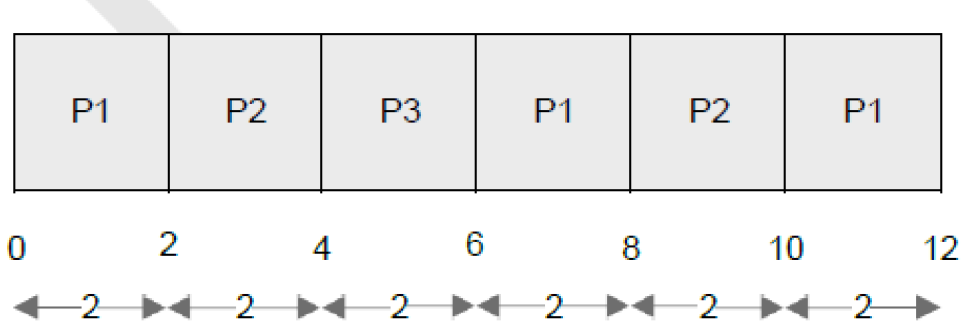


. Round Robin Scheduling

- Each process in the 'Ready' queue is executed for a pre-defined time slot.
- The execution starts with picking up the first process in the 'Ready' queue. It is executed for a pre-defined time
- When the pre-defined time elapses or the process completes (before the pre- defined time slice), the next process in the 'Ready' queue is selected for execution.
- This is repeated for all the processes in the 'Ready' queue
- Once each process in the 'Ready' queue is executed for the pre-defined time period, the scheduler comes back and picks the first process in the 'Ready' queue again for execution.
- Round Robin scheduling is similar to the FCFS scheduling and the only difference is that a time slice based preemption is added to switch the execution between the processes in the 'Ready' queue

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 6, 4, 2 milliseconds respectively, enters the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in RR algorithm with Time slice= 2ms.

Solution: The scheduler sorts the '*Ready*' queue based on the FCFS policy and picks up the first process P1 from the 'Ready' queue and executes it for the time slice 2ms. When the time slice is expired, P1 is preempted and P2 is scheduled for execution. The Time slice expires after 2ms of execution of P2. Now P2 is preempted and P3 is picked up for execution. P3 completes its execution within the time slice and the scheduler picks P1 again for execution for the next time slice. This procedure is repeated till all the processes are serviced.



The order in which the processes are scheduled for execution is represented as
The waiting time for all the processes are given as
Waiting Time for P1 = 0 + (6-2) + (10-8) = 0+4+2= 6ms (P1 starts executing first and waits for two time slices to get execution back and again 1 time slice for getting CPU time)
Waiting Time for P2 = (2-0) + (8-4) = 2+4 = 6ms (P2 starts executing after P1 executes for 1 time slice and waits for two time slices to get the CPU time)
Waiting Time for P3 = (4 -0) = 4ms (P3 starts executing after completing the first time slices for P1 and P2 and completes its execution in a single time slice.)
Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P1+P2+P3)) / 3
= (6+6+4)/3 = 16/3
= 5.33 milliseconds
Turn Around Time (TAT) for P1 = 12 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 10 ms (-Do-)
Turn Around Time (TAT) for P3 = 6 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P1+P2+P3)) / 3
= (12+10+6)/3 = 28/3
= 9.33 milliseconds.

Preemptive scheduling – Priority based Scheduling

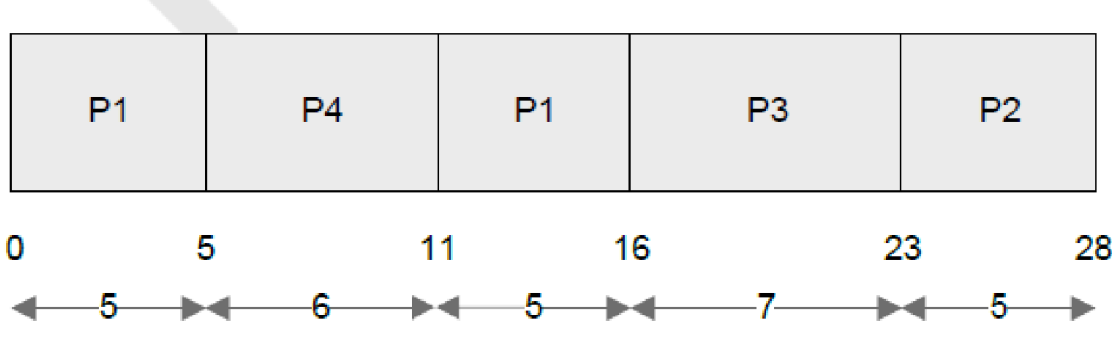
- Same as that of the ***non-preemptive priority*** based scheduling except for the switching of execution between tasks

- In ***preemptive priority*** based scheduling, any high priority process entering the 'Ready' queue is immediately scheduled for execution whereas in the ***non-preemptive*** scheduling any high priority process entering the 'Ready' queue is scheduled only after the currently executing process completes its execution or only when it voluntarily releases the CPU

❖ The priority of a task/process in preemptive priority based scheduling is indicated in the same way as that of the mechanisms adopted for non- preemptive multitasking.

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 1, 3, 2 (0- highest priority, 3 lowest priority) respectively enters the ready queue together. A new process P4 with estimated completion time 6ms and priority 0 enters the 'Ready' queue after 5ms of start of execution of P1. Assume all the processes contain only CPU operation and no I/O operations are involved.

Solution: At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the scheduler picks up the process with the highest priority (In this example P1 with priority 1) for scheduling. Now process P4 with estimated execution completion time 6ms and priority 0 enters the '*Ready*' queue after 5ms of start of execution of P1. The processes are re-scheduled for execution in the following order



The waiting time for all the processes are given as
Waiting Time for P1 = 0 + (11-5) = 0+6 =6 ms (P1 starts executing first and gets
Preempted by P4 after 5ms and again gets the CPU time after completion of P4)
Waiting Time for P4 = 0 ms (P4 starts executing immediately on entering the
'Ready' queue, by preempting P1)

Waiting Time for P3 = 16 ms (P3 starts executing after completing P1 and P4) Waiting Time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3) Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P1+P4+P3+P2)) / 4
= (6 + 0 + 16 + 23)/4 = 45/4
= 11.25 milliseconds

Turn Around Time (TAT) for P1 = 16 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 6ms (Time spent in Ready Queue + Execution Time
= (Execution Start Time – Arrival Time) + Estimated Execution Time = (5-5) + 6 = 0 + 6)
Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time) Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time) Average Turn Around Time= (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P2+P4+P3+P1)) / 4
= (16+6+23+28)/4 = 73/4
= 18.25 milliseconds

Timing Requirements on Processes

Processes can have several different types of timing requirements imposed on them by the application.The timing requirements on a set of processes strongly influence the type of scheduling that is appropriate.A scheduling policy must define the timing requirements that it uses to determine whether a schedule is valid. Before studying scheduling proper, we outline the types of process timing requirements that are useful in embedded system design.

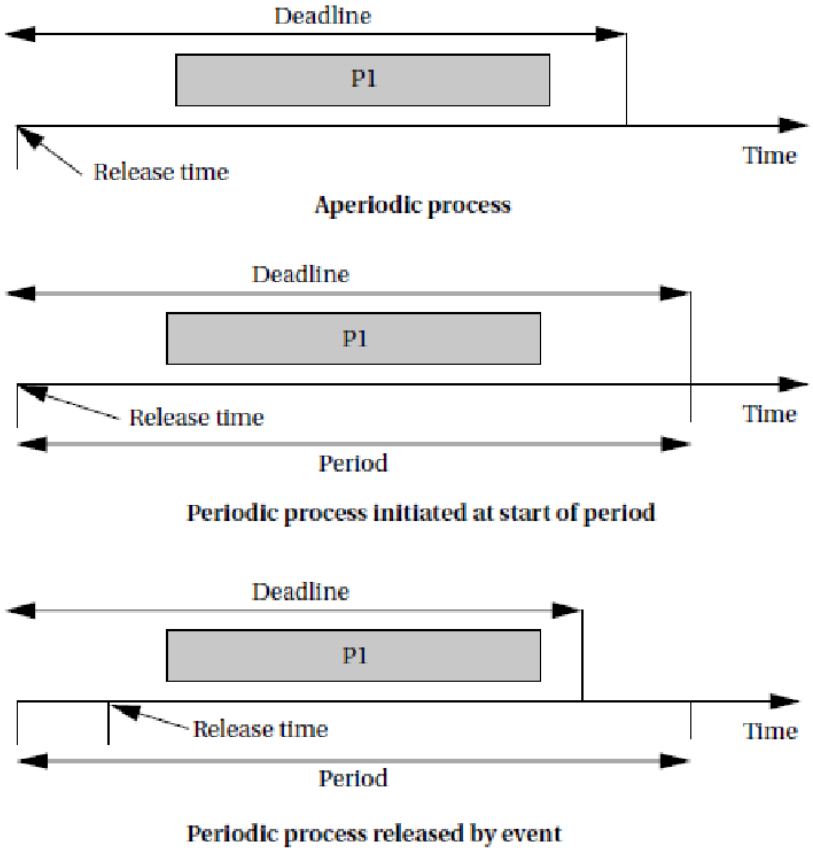


Figure illustrates different ways in which we can define two important requirements on processes: ***release time*** and ***deadline***.

The release time is the time at which the process becomes ready to execute; this is not necessarily the time at which it actually takes control of the CPU and starts to run.

An aperiodic process is by definition initiated by an event, such as external data arriving or data computed by another process.

The release time is generally measured from that event, although the system may want to make the process ready at some interval after the event itself.

For a periodically executed process, there are two common possibilities. In simpler systems, the process may become ready at the beginning of the period.

More sophisticated systems, such as those with data dependencies between processes, may set the release time at the arrival time of certain data, at a time after the start of the period.

A deadline specifies when a computation must be finished. The deadline for an aperiodic process is generally measured from the release time, since that is the only reasonable time reference. The deadline for a periodic process may in genera occur at some time other than the end of the period.

Rate requirements are also fairly common. A rate requirement specifies how quickly processes must be initiated.

The ***period*** of a process is the time between successive executions. For example, the period of a digital filter is defined by the time interval between successive input samples.
The process's ***rate*** is the inverse of its period. In a multirate system, each process executes at its own distinct rate. The most common case for periodic processes is for the initiation interval to be equal to the period. However,pipelined execution of processes allows the initiation interval to be less than the period.

CPU Metrics

We also need some terminology to describe how the process actually executes.
The ***initiation time*** is the time at which a process actually starts executing on the CPU.
The ***completion time*** is the time at which the process finishes its work.
The most basic measure of work is the amount of ***CPU time*** expended by a process. The CPU time of process i is called T_i .
The total CPU time consumed by a set of processes is

$$T = \sum_{1 \leq i \leq n} T_i.$$

We need a basic measure of the efficiency with which we use the CPU. The simplest and most direct measure is utilization:

$$U = \frac{\text{CPU time for useful work}}{\text{total available CPU time}}.$$

Utilization is the ratio of the CPU time that is being used for useful computations to the total available CPU time. This ratio ranges between 0 and 1 with 1 meaning that all the available CPU time is being used for system purposes over an interval of time t, then the CPU utilization is

$$U = \frac{T}{t}.$$

When we evaluate the utilization of the CPU, we generally do so over a finite period that covers all possible combinations of process executions. For periodic processes, the length of time that must be considered is the ***hyperperiod***, which is the least-common multiple of the periods of all the processes. (The complete schedule for the least-common multiple of the periods is sometimes called the ***unrolled schedule***.) If we evaluate the hyperperiod,we are sure to have considered all possible combinations of the periodic processes.

Utilization of a set of processes

We are given three processes, their execution times, and their periods:

Process	Period	Execution time
P1	1.0×10^{-3}	1.0×10^{-4}
P2	1.0×10^{-3}	2.0×10^{-4}
P3	5.0×10^{-3}	3.0×10^{-4}

The least common multiple of these periods is 5×10^{-3} s.

In order to calculate the utilization, we have to figure out how many times each process is executed in one hyperperiod: P1 and P2 are each executed five times while P3 is executed once.

We can now determine the utilization over the hyperperiod:

$$U = \frac{5.1 \times 10^{-4} + 5.2 \times 10^{-4} + 1.3 \times 10^{-4}}{5 \times 10^{-3}} = 0.36$$

This is well below our maximum utilization of 1.0.

One very simple scheduling policy is known as ***cyclostatic*** scheduling or sometimes as ***Time Division Multiple Access*** scheduling.

A cyclostatic schedule is divided into equal-sized time slots over an interval equal to the length of the hyperperiod *H*.

Processes always run in the same time slot. Two factors affect utilization:the number of time slots used and the fraction of each time slot that is used for useful work.

Depending on the deadlines for some of the processes,we may need to leave some time slots empty.And since the time slots are of equal size,some short processes may have time left over in their time slot.

We can use utilization as a schedulability measure: the total CPU time of all the processes must be less than the hyperperiod.

Another scheduling policy that is slightly more sophisticated is ***round robin***.

round robin uses the same hyperperiod as does cyclostatic.

It also evaluates the processes in order.

But unlike cyclostatic scheduling,if a process does not have any useful work to do, the round-robin scheduler moves on to the next process in order to fill the time slot with useful work. In this example, all three processes execute during the first hyperperiod, but during the second one, *P*1 has no useful work and is skipped. The processes are always evaluated in the same order.The last time slot in the hyperperiod is left empty;if we have occasional, non-periodic tasks without deadlines, we can execute them in these empty time slots.

Round-robin scheduling is often used in hardware such as buses because it is very simple to implement but it provides some amount of flexibility.

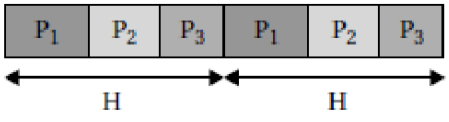


FIGURE 6.7
Cyclostatic scheduling.

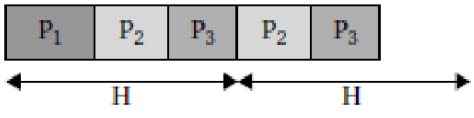


FIGURE 6.8
Round-robin scheduling.

Priority Based Scheduling

After assigning priorities, the OS takes care of the rest by choosing the highest-priority ready process. There are two major ways to assign priorities: ***static*** priorities that do not change during execution and ***dynamic*** priorities that do change.

Rate-Monotonic Scheduling

Rate monotonic scheduling is a priority algorithm that belongs to the static priority scheduling category of Real Time Operating Systems.
It is preemptive in nature.
The priority is decided according to the cycle time of the processes that are involved. If the process has a small job duration, then it has the highest priority.
Thus, if a process with highest priority starts execution, it will preempt the other running processes. The priority of a process is inversely proportional to the period it will run for.
A set of processes can be scheduled only if they satisfy the following equation:

$$\sum_{k=1}^{n} \frac{Ci}{Ti} \leq U = n\left(2^{1/n} - 1\right)$$

Where n is the number of processes in the process set, C_i is the computation time of the process, T_i is the Time period for the process to run and U is the processor utilization.

Example:

An example to understand the working of the Rate monotonic scheduling algorithm.

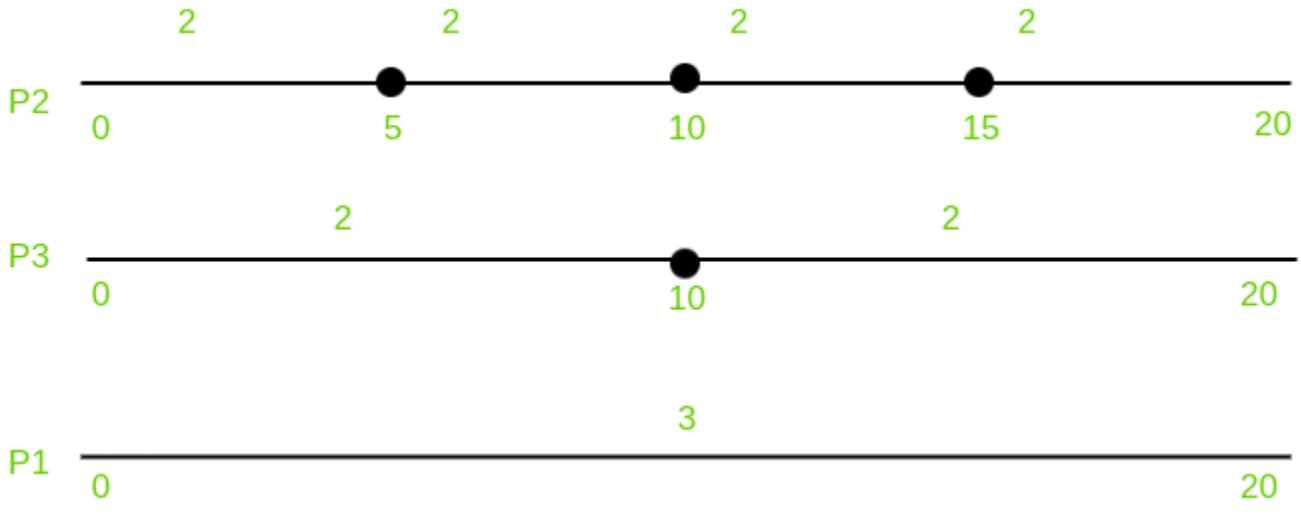
Processes	Execution Time (C)	Time period (T)
P1	3	20
P2	2	5
P3	2	10

$$n(2^{1/n} - 1) = 3 (2^{1/3} - 1) = 0.7977$$

$$U = 3/20 + 2/5 + 2/10 = 0.75$$

It is less than 1 or 100% utilization. The combined utilization of three processes is less than the threshold of these processes that means the above set of processes are schedulable and thus satisfies the above equation of the algorithm.

1. **Scheduling time –**
   For calculating the Scheduling time of algorithm we have to take the LCM of the Time period of all the processes. LCM (20, 5, 10) of the above example is 20. Thus we can schedule it by 20 time units.
2. **Priority –**
   As discussed above, the priority will be the highest for the process which has the least running time period. Thus P2 will have the highest priority, after that P3 and lastly P1.
   P2 > P3 > P1
3. **Representation and flow –**



Above figure says that, Process P2 will execute two times for every 5 time units, Process P3 will execute two times for every 10 time units and Process P1 will execute three times in 20 time units.

- Process P2 will run first for 2 time units because it has the highest priority. After completing its two units, P3 will get the chance and thus it will run for 2 time units.
- As we know that process P2 will run 2 times in the interval of 5 time units and process P3 will run 2 times in the interval of 10 time units, they have fulfilled the criteria and thus now process P1 which has the least priority will get the chance and it will run for 1 time. And here the interval of five time units have completed. Because of its priority P2 will preempt P1 and thus will run 2 times. As P3 have completed its 2 time units for its interval of 10 time units, P1 will get chance and it will run for the remaining 2 times, completing its execution which was thrice in 20 time units.
- Now 9-10 interval remains idle as no process needs it. At 10 time units, process P2 will run for 2 times completing its criteria for the third interval (10-15). Process P3 will now run for two times completing its execution. Interval 14-15 will again remain idle for the same reason mentioned above. At 15 time unit, process P2 will execute for two times completing its execution.This is how the rate monotonic scheduling works.

Conditions :

The analysis of Rate monotonic scheduling assumes few properties that every process should possess. They are :

1. Processes involved should not share the resources with other processes.
2. Deadlines must be similar to the time periods. Deadlines are deterministic.
3. Process running with highest priority that needs to run, will preempt all the other processes.
4. Priorities must be assigned to all the processes according to the protocol of Rate monotonic scheduling.

Advantages :

1. It is easy to implement.
2. If any static priority assignment algorithm can meet the deadlines then rate monotonic scheduling can also do the same. It is optimal.
3. It consists of calculated copy of the time periods unlike other time-sharing algorithms as Round robin which neglects the scheduling needs of the processes.

Disadvantages :

1. It is very difficult to support aperiodic and sporadic tasks under RMA.
2. RMA is not optimal when tasks period and deadline differ.

Earliest Deadline First (EDF) CPU scheduling algorithm

- **Earliest Deadline First (EDF)** is an optimal dynamic priority scheduling algorithm used in real-time systems.
  It can be used for both static and dynamic real-time scheduling.
- EDF uses priorities to the jobs for scheduling. It assigns priorities to the task according to the absolute deadline. The task whose deadline is closest gets the highest priority. The priorities are assigned and changed in a dynamic fashion.
- EDF is very efficient as compared to other scheduling algorithms in real-time systems. It can make the CPU utilization to about 100% while still guaranteeing the deadlines of all the tasks.
- EDF includes the kernel overload. In EDF, if the CPU usage is less than 100%, then it means that all the tasks have met the deadline. EDF finds an optimal feasible schedule. The feasible schedule is one in which all the tasks in the system are executed within the deadline. If EDF is not able to find a feasible schedule for all the tasks in the real-time system, then it means that no other task scheduling algorithms in real-time systems can give a feasible schedule. All the tasks which are ready for execution should announce their deadline to EDF when the task becomes runnable.
- EDF scheduling algorithm does not need the tasks or processes to be periodic and also the tasks or processes require a fixed CPU burst time. In EDF, any executing task can be preempted if any other periodic instance with an earlier deadline is ready for

execution and becomes active. Preemption is allowed in the Earliest Deadline First scheduling algorithm.

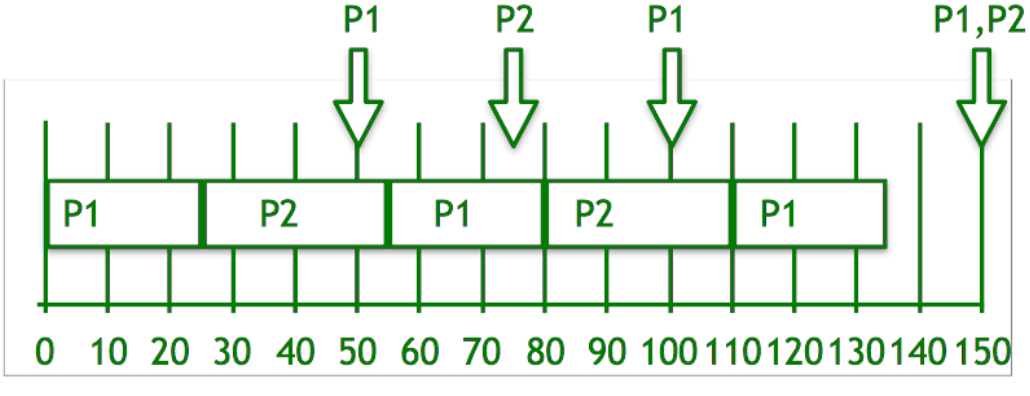
Example:

Consider two processes P1 and P2.

Processes	Period	Execution Time
P1	50	25
P2	75	30

0 25

P1	P2						



Steps for solution:

1. Deadline pf P1 is earlier, so priority of P1>P2.
2. Initially P1 runs and completes its execution of 25 time.
3. After 25 times, P2 starts to execute until 50 times, when P1 is able to execute.
4. Now, comparing the deadline of (P1, P2) = (100, 75), P2 continues to execute.
5. P2 completes its processing at time 55.

6. P1 starts to execute until time 75, when P2 is able to execute.
7. Now, again comparing the deadline of (P1, P2) = (100, 150), P1 continues to execute.
8. Repeat the above steps…
9. Finally at time 150, both P1 and P2 have the same deadline, so P2 will continue to execute till its processing time after which P1 starts to execute.

Limitations of EDF scheduling algorithm:

- Transient Overload Problem
- Resource Sharing Problem
- Efficient Implementation Problem