O.S 2,3,4,5 units — Musharaf

①

UNIT- 2

## ROCESSOR AND USER MODE [OR] DUAL OPERATIONS OF OS [OR] USER MODE AND KERNEL MODE

Modern Operating Systems have two basic modes in which they can execute a certain program – User Mode and Kernel Mode.
The Processor (CPU) switches between these two modes.
Applications run in user mode , and core OS components run in kernel mode.
These modes define standardised instructions for deciding what resources can be accessed.
The dual mode protect data and provide system security.

**USER MODE:**

It has restricted access to the resources.
CPU has restrictions , therefore it can have only access to limited instructions and memory.
Utility applications such as text editors, media players are run in the user mode.
User mode does not have direct access to the computer hardware.
The mode bit of user mode is "1".
When an application or program is executed, its initial state and operation mode are loaded on stack. At this point, Processor starts executing the program in this mode.
For perfroming hardware related tasks or whether the user applications requets for a service from the OS, or if any interrrupt occurs – the system must switch to kernel mode. CPU (processor) do the esssential tasks by stroring the current state of the user program to stack again. After completing it, CPU resumes the program where it left.

**KERNEL MODE:**

Kernel mode also called as Supervisor mode or system mode or Privileged mode.
This mode has full access to memory, I/O and other resources. In this mode, CPU can execute any instructions and have full access to underlying hardware.
The core functionalities of the OS always run in kernel mode.
The processes have full rights to access resources, allowing them read/write to the storage media.
All the bottom(low) level tasks of the OS are perfromed in kernel mode.
Kernel mode handles all the processes which requires hardware support.
So all the processes and instructions that the user is restricted are executed in kernel mode of the OS.
The mode bit of the kernel mode is "0".

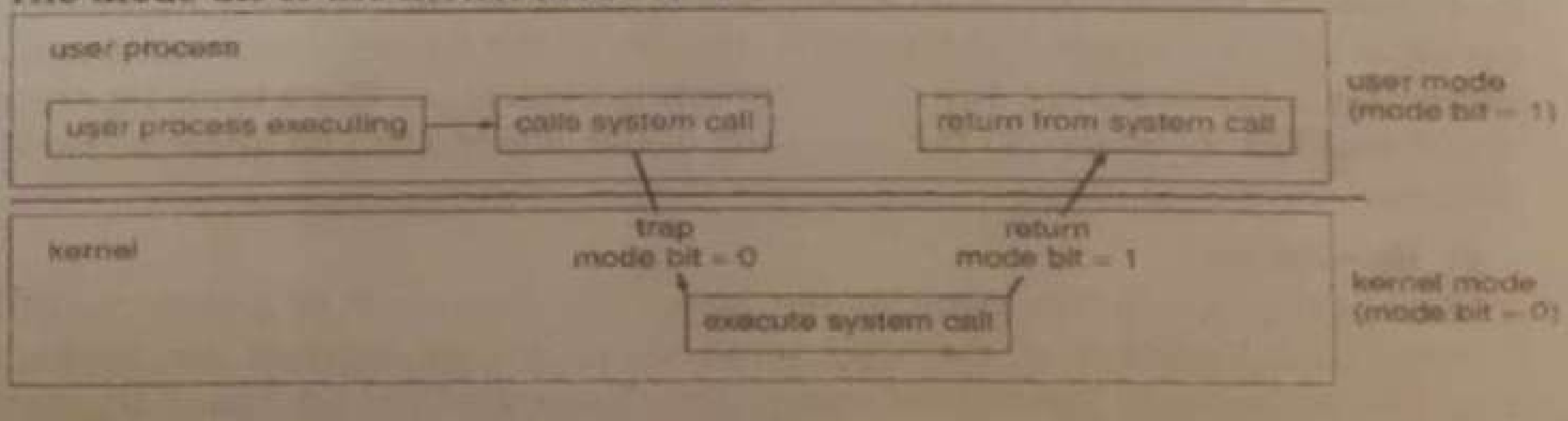


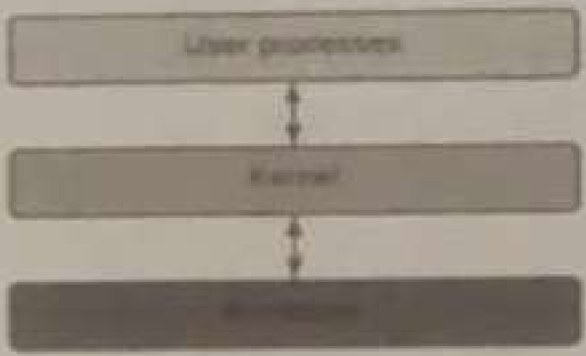
Figure 1.10 Transition from user to kernel mode.

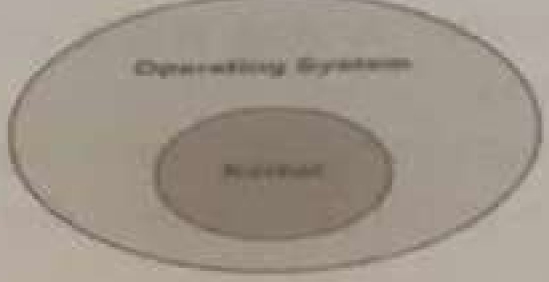
# KERNEL

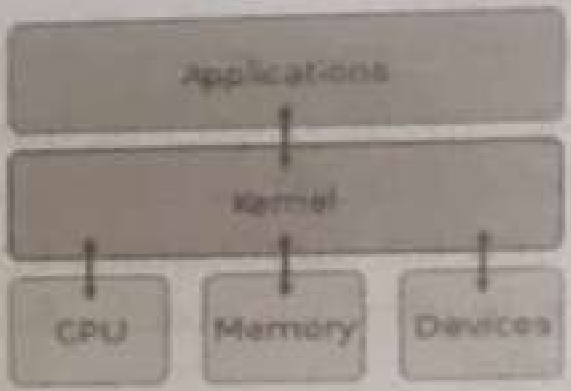
**Kernel** is the important part of an Operating System. The kernel is the first progra that is loaded after the boot loader whenever we start a system. The Kernel is presen in the memory until the Operating System is shut-down.

Kernel provides an interface between the user and the hardware components of the system. When a process makes a request to the Kernel, then it is called System Call.

A Kernel is a computer program that is the heart and core of an Operating System. Since the Operating System has control over the system so, the Kernel also has control over everything in the system. It is the most important part of an Operating System. Whenever a system starts, the Kernel is the first program that is loaded after the bootloader because the Kernel has to handle the rest of the thing of the system for the Operating System.







## Functions of Kerenl:

**Access Computer Resource** – A Kernel accesses various computer resources like the CPU, I/O devices and other resources. Kernel is present in between the user and the resources of the system to establish the communication.

**Resource Management** – Kernel shares the resources between various processes in a way that there is uniform access to the resources by every process.

**Memory Management** – Generally memory management is done by the kernel because every process needs some memory space and memory has to be allocated and deallocated for its execution.

**Device Management** – The allocation of peripheral devices connected in the system used by the processes is managed by the kernel.

## Types of Kernel:

The different types of kernels are as follows:

### a. Monolithic Kernels

In monolithic Kernels both user services and the kernel services are implemented in the same memory space. By doing this, the size of the Kernel is increased and at the same time it increases the size of the Operating System. As there is no separate User Space and Kernel Space, so the execution of the process will be faster in Monolithic Kernels.

### b. Microkernel

A Microkernel is not the same as the Monolithic kernel. It is a little bit different because in a Microkernel, the user services and kernel services are implemented into different spaces. Because of using User Space and Kernel Space separately, it reduces the size of the Kernel and in turn, reduces the size of the Operating System.

As we are using different spaces for user and kernel service, the communication between application and services is done with the help of message parsing because of this it reduces the speed of execution.

The advantage of microkernel is that it can easily add new services at any time.

The disadvantage of microkernel is that here we are using User Space and Kernel Space separately. So, the communication between these can reduce the overall execution time.

**c. Hybrid Kernel**

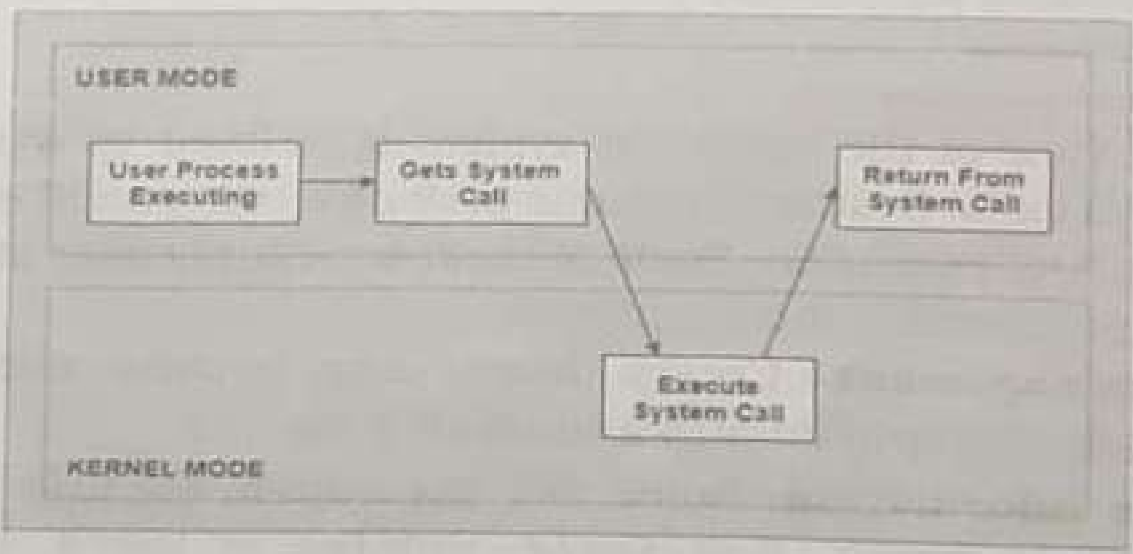
It is the combination of both Monolithic Kernel and Microkernel. It uses the speed of the Monolithic Kernel and the modularity of Microkernel.

Hybrid kernels are micro kernels having some non-essential code in kernel-space in order for the code to run more quickly than it would be in user-space. So, some services like network stack or file systems are run in Kernel space to reduce the performance overhead, but still, it runs kernel code as servers in the user-space.

******

## System Calls

A **system call** is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS.
System call offers the services of the operating system to the user programs via API (Application Programming Interface). System calls are the only entry points for the kernel system.
A system call is a way for a user program to interface with the operating system. The program requests several services, and the OS responds by invoking a series of system calls to satisfy the request. System calls are predefined functions that the operating system may directly invoke if a high-level language is used.
The **Application Program Interface (API)** connects the operating system's functions to user programs. It acts as a link between the operating system and a process, allowing user-level programs to request operating system services. The kernel system can only be accessed using system calls. System calls are required for any programs that use resources.
The interface between a process and an operating system is provided by system calls. System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call.



## Types of System Calls:

There are mainly five types of system calls. These are explained in detail as follows:

Process Control:

These system calls deal with processes such as process creation, process termination etc.

File Management:

These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.

Device Management:

These system calls are responsible for device manipulation such as reading from device buffers, writing into device buffers etc.

Information Maintenance:

These system calls handle information and its transfer between the operating system and the user program.

Communication:

These system calls are useful for interprocess communication. They also deal with creating and deleting a communication connection.

Some of the examples of all the above types of system calls in Windows and Unix are given as follows –

| Types of System Calls | Windows | Linux |
|---|---|---|
| Process Control | CreateProcess()ExitProcess()WaitForSingleObject() | fork()exit()wait() |
| File Management | CreateFile()ReadFile()WriteFile()CloseHandle() | open()read()write()close() |
| Device Management | SetConsoleMode()ReadConsole()WriteConsole() | ioctl()read()write() |
| Information Maintenance | GetCurrentProcessID()SetTimer()Sleep() | getpid()alarm()sleep() |
| Communication | CreatePipe()CreateFileMapping()MapViewOfFile() | pipe()shmget()mmap( |

*******

imp

## System Programs

System programs provide a convenient environment for program development and execution. The can be divided into:

- File manipulation
- Status information
- File modification
- Programming language support
- Program loading and execution
- Communications
- Application programs

Most users' view of the operation system is defined by system programs, not the actual system calls. System Programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex.

a. **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

b. **Status information:** Some ask the system for info - date, time, amount of available memory, disk space, number of users. Others provide detailed performance, logging, and debugging information.

Typically, these programs format and print the output to the terminal or other output devices. Some systems implement a registry - used to store and retrieve configuration information

c. **File modification:** Text editors to create and modify files. Special commands to search contents of files or perform transformations of the text.
d. **Programming-language support:** - Compilers, assemblers, debuggers and interpreters sometimes provided.
e. **Program loading and execution:**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language.
f. **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems. Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another.

******

## System View of the Process and Resources

System View:

The OS may also be viewed as just a resource allocator. A computer system comprises various sources, such as hardware and software, which must be managed effectively. The operating system manages the resources, decides between competing demands, controls the program execution, etc. According to this point of view, the operating system's purpose is to maximize performance. The operating system is responsible for managing hardware resources and allocating them to programs and users to ensure maximum performance.

The hardware interacts with the operating system than with the user from a system viewpoint. The hardware and the operating system interact for a variety of reasons, including:

Resource Allocation:

The hardware contains several resources like registers, caches, RAM, ROM, CPUs, I/O interaction, etc. These are all resources that the operating system needs when an application program demands them. Only the operating system can allocate resources, and it has used several tactics and strategies to maximize its processing and memory space. The operating system uses a variety of strategies to get the most out of the hardware resources, including paging, virtual memory, caching, and so on. These are very important in the case of various user viewpoints because inefficient resource allocation may affect the user viewpoint, causing the user system to lag or hang, reducing the user experience.

Control Program:

The control program controls how input and output devices (hardware) interact with the operating system. The user may request an action that can only be done with I/O devices; in this case, the operating system must also have proper communication, control, detect, and handle such devices.

******

## Process Abstraction

Abstractions provide an **interface** to application programmers that separate **policy**—what the interface commits to accomplishing—from **mechanism**—how the interface is implemented.

1. Example Abstraction: File

What **undesirable properties** do file systems hide?

- Disks are slow!
- Chunks of storage are actually distributed all over the disk.
- Disk storage may fail!

What **new capabilities** do files add?

- Growth and shrinking.
- Organization into directories.

What **information** do files help organize?

- Ownership and permissions.
- Access time, modification time, type, etc.

2. Preview of Coming Abstractions

- **Threads** abstract the CPU.
- **Address spaces** abstract memory.
- **Files** abstract the disk.
- We will return to these abstractions. We are starting with an organizing principle.

3. The Process:

**Processes are the most fundamental operating system abstraction.**

- Processes organize information about other abstractions and represent a single thing that the computer is "doing."
- You know processes as app(lication)s.

4. Organizing Information:

Unlike threads, address spaces and files, processes are **not tied to a hardware component**. Instead, they contain other abstractions.

Processes contain:

- one or more **threads**,
- an **address space**, and
- zero or more open **file handles** representing files.

........

## Process Hierarchy

Now-a-days all general purpose operating systems permit a user to create and destroy processes. A process can create several new processes during its time of execution.

The creating process is called the Parent Process and the new process is called Child Process.

There are different ways for creating a new process. These are as follows –

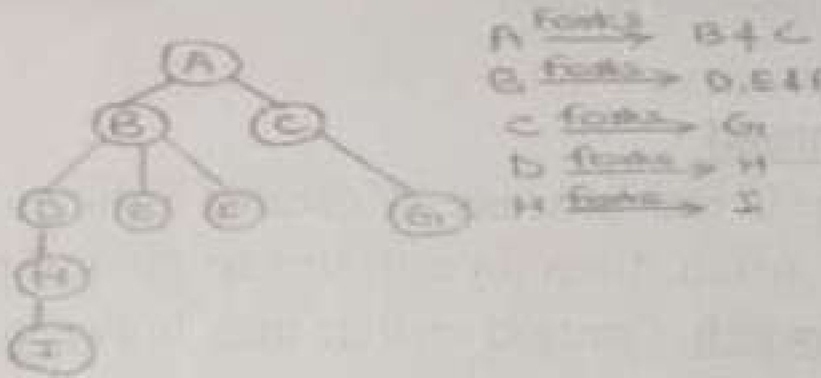
**Execution** – The child process is executed by the parent process concurrently or it waits till all children get terminated.

**Sharing** – The parent or child process shares all resources like memory or files or children process shares a subset of parent's resources or parent and children process share no resource in common.

The reasons that parent process terminates the execution of one of its children are as follows –

- The child process has exceeded its usage of resources that have been allocated. Because of this there should be some mechanism which allows the parent process to inspect the state of its children process.
- The task that is assigned to the child process is no longer required.

Let us discuss this with an example
In unix this is done by the 'Fork' system call, which creates a 'child' process and the 'exit system call', which terminates current process.



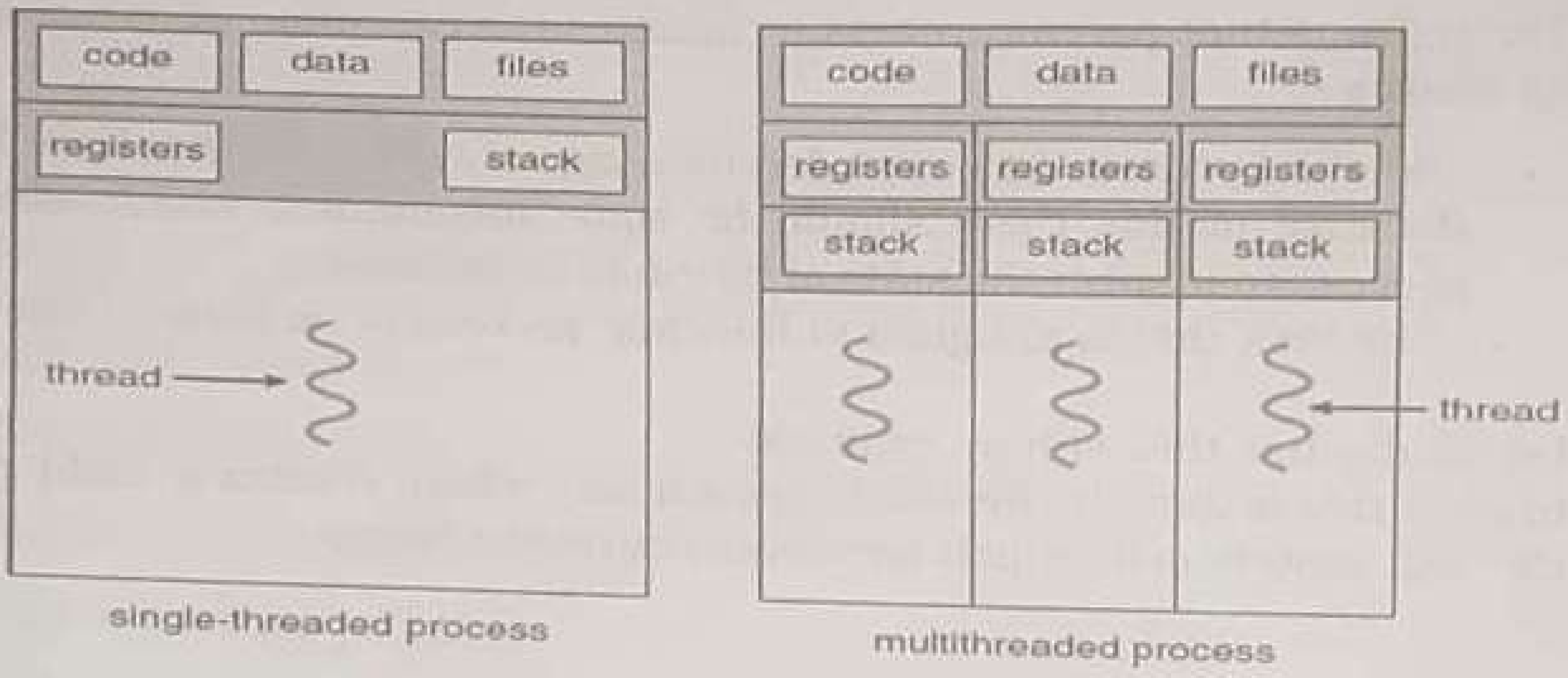
The root of tree is a special process created by the operating system during startup.
**A process can choose to wait for children to terminate.**
**Example C issued a wait() system call it would block until G-Finished.**

## THREADS

- A thread is referred as a light-weight process, also called mini-process.
- A traditional process (heavy-weight process) has a "single-thread" of control.
- Most modern operating systems provide features which enable a process that should contain "Multiple-threads of control".
- If a process consists of one or more threads of control, it can perform more than one task at a time.
- Threads share the same address space and global variables of a process to which they belongs. Also all the threads of a process share the code section, data section and OS resources.
- A thread contains a thread ID, a program counter, a register set and a stack.
- Threads are a popular way to improve application performance.
- Thread is equivalent to a classical process.
- Each thread belongs to exactly one process, and no thread can exist outside a process.
- Each thread represents a separate flow of control.

- Threads have been successfully used in implementing network servers, web server.
- For example: a web browser might have one thread to display images, another thread for text and another thread retrieves data from the network etc.

The following figure shows the working of a single-threaded and a multithreaded process.



## Advantages of Thread:

1. **Responsiveness:** If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.

2. **Faster context switch:** Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.

3. **Effective utilization of multiprocessor system:** If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.

4. **Resource sharing:** Resources like code, data, and files can be shared among all threads within a process.
Note: stack and registers can't be shared among the threads. Each thread has its own stack and registers.

5. **Communication:** Communication between multiple threads is easier, as the threads shares common address space. While in process we have to follow some specific communication technique for communication between two process.

6. **Enhanced throughput of the system:** If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

## Threading Issues

### Threading Issues in OS:

- fork() and exec() System Calls
- Thread Cancellation
- Signal Handling
- Thread Pool
- Thread Specific Data

### 1. fork() and exec() System Calls:

The fork() and exec() are the system calls. The fork() call creates a duplicate process of the process that invokes fork(). The new duplicate process is called child process and process invoking the fork() is called the parent process. Both the parent process and the child process continue their execution from the instruction that is just after the fork().

Let us now discuss the issue with the fork() system call. Consider that a thread of the multithreaded program has invoked the fork(). So, the fork() would create a new duplicate process. Here the issue is whether the new duplicate process created by fork() will duplicate all the threads of the parent process or the duplicate process would be single-threaded.

Well, there are two versions of fork() in some of the UNIX systems. Either the fork() can duplicate all the threads of the parent process in the child process or the fork() would only duplicate that thread from parent process that has invoked it.

Which version of fork() must be used totally depends upon the application.

Next system call i.e. exec() system call when invoked replaces the program along with all its threads with the program that is specified in the parameter to exec(). Typically the exec() system call is lined up after the fork() system call.

Here the issue is if the exec() system call is lined up just after the fork() system call then duplicating all the threads of parent process in the child process by fork() is useless. As the exec() system call will replace the entire process with the process provided to exec() in the parameter.

In such case, the version of fork() that duplicates only the thread that invoked the fork() would be appropriate.

### 2. Thread cancellation:

Termination of the thread in the middle of its execution it is termed as 'thread cancellation'. Let us understand this with the help of an example. Consider that there is a multithreaded program which has let its multiple threads to search through a database for some information. However, if one of the thread returns with the desired result the remaining threads will be cancelled.

Now a thread which we want to cancel is termed as target thread. Thread cancellation can be performed in two ways:

**Asynchronous Cancellation:** In asynchronous cancellation, a thread is employed to terminate the target thread instantly.

**Deferred Cancellation:** In deferred cancellation, the target thread is scheduled to check itself at regular interval whether it can terminate itself or not.

The issue related to the target threads are listed below:

- What if the resources had been allotted to the cancel target thread?
- What if the target thread is terminated when it was updating the data, it w sharing with some other thread.
- Here the **asynchronous cancellation** of the thread where a thread immediately cancels the target thread without checking whether it is holding any resources or not creates troublesome.
- However, in **deferred cancellation**, the thread that indicates the target thread about the cancellation, the target thread crosschecks its flag in order to confirm that it should it be cancelled immediately or not.

**3. Signal Handling:**

Signal handling is more convenient in the single-threaded program as the signal would be directly forwarded to the process. But when it comes to multithreaded program, the issue arrives to which thread of the program the signal should be delivered.

Generally, signal is used in UNIX systems to notify a process that a particular event has occurred. A signal received either synchronously or asynchronously, based on the source of and the reason for the event being signalled.

In UNIX systems, a signal is used to notify a process that a particular event has happened. Based on the source of the signal, signal handling can be categorized as:

**Asynchronous Signal:** The signal which is generated outside the process which receives it.

**Synchronous Signal:** The signal which is generated and delivered in the same process.

All signals, whether synchronous or asynchronous, follow the same pattern as given below –

- A signal is generated by the occurrence of a particular event.
- The signal is delivered to a process.
- Once delivered, the signal must be handled.

**4. Thread Pool:**

When a user requests for a webpage to the server, the server creates a separate thread to service the request. Although the server also has some potential issues. Consider if we do not have a bound on the number of actives thread in a system and would create a new thread for every new request then it would finally result in exhaustion of system resources.

The solution to this issue is the **thread pool**. The idea is to create a finite amount of threads when the process starts. This collection of threads is referred to as the thread pool. The threads stay in the thread pool and wait till they are assigned any request to be serviced.

Whenever the request arrives at the server, it invokes a thread from the pool and assigns it the request to be serviced. The thread completes its service and return back to the pool and wait for the next request.

**5. Thread-Specific data:**

We all are aware of the fact that the threads belonging to the same process share the data of that process. Here the issue is what if each particular thread of the process needs its own copy of data. So the specific data associated with the specific thread is referred to as **thread-specific data.**

Consider a transaction processing system, here we can process each transaction in a different thread. To determine each transaction uniquely we will associate a unique identifier with it. Which will help the system to identify each transaction uniquely. As we are servicing each transaction in a separate thread. So we can use thread-specific data to associate each thread to a specific transaction and its unique id.

*******

## Thread Libraries

**Thread Libraries** has a collection of functions that useful in creating and controlling threads. Programmers can access these thread libraries using an application programming interface (API). Thread libraries can be the *user level library* or *kernel level library.*

A thread library provides the programmer an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main thread libraries are in use today:

- POSIX Pthreads,
- Win32 Library
- Java Library

### Pthread Library:

Pthreads are also termed as **POSIX** thread library. This can be implemented either at the *userspace* or at the *kernel space*. Pthreads library is often implemented at LINUX, UNIX, Solaris, Mac OSX. The Pthread program must always have a **pthread.h** header file.

### Win32 Library:

Creation of thread in Win2 library is similar to pthread library. To create a thread using the Win32 library always include **windows.h** header file in the program. The Win32 thread library is a *kernel-level* library which means invoking the Win32 library function results in a system call.

### Java Thread Library:

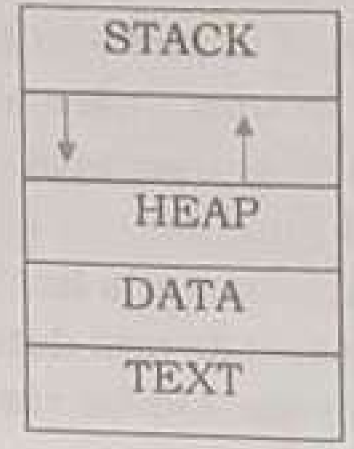
- You must have seen that mostly the java virtual machine JVM runs on the top of the host operating system. That's why java threads are created and controlled by using the available library at the host operating system.
- Therefore, in the Windows operating system, the java threads are implemented using Win32 API and in operating systems such as Linux and UNIX, the java thread is implemented using Pthread library.

*****

## PROCESS

- Process is a program in execution.
- A process is a unit of work in a system.
- A computer system consists of a collection of processes, they are "operating system-processes" which execute systems code and "user processes" which execute user code.
- A compiler, word processing program, web browser, sending output to a printer is the examples for a process.

### STRUCTURE OF A PROCESS IN MEMORY



a. Stack: It contains temporary data such as local variables, function parameters and return addresses.
b. Heap: It is a memory dynamically allocated during process runtime.
c. Data Section: It contains global variables.
d. Text Section: The program code of a process.

### PROCESS STATE

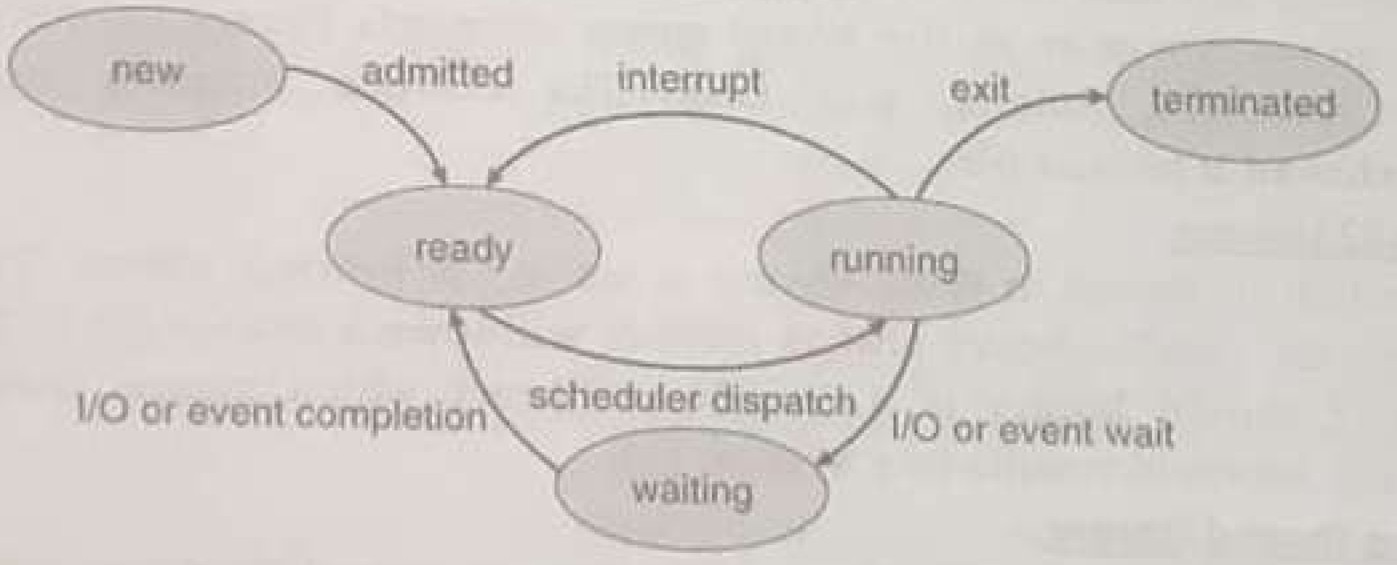


**Diagram of a Process State**

As a process executes, it changes states. The state of a process is defined by the current activity of that process. Each process may be in one of the following states:

a) <u>New</u>: If the process is in the new state, it means that the process is being created. In this case only memory is allocated, but not CPU.

b) <u>Ready</u>: A process is said to be ready if it is waiting to be assigned to a processor (CPU), so that it could use a CPU. In ready state the program is waiting for CPU in ready queue (Ready list) .The assignment of the CPU to the first process on the ready list is called "<u>Dispatching</u>".

c) <u>Running</u>: A process is said to be running if it is has the CPU (processor) and the instructions are being executed.

d) <u>Waiting</u>: A process is said to be waiting if it is waiting for an event to occur, such as an I/O completion. Until the event is completed, the process cannot proceed further.

e) <u>Terminated</u>: If a process enters the terminated state, it means that the process has finished execution of its tasks (jobs).

......

<u>PROCESS CONTROL BLOCK</u> [PCB]

| |
|---|
| Process State |
| Program Counter |
| Register Contents |
| Memory Limits |
| CPU Utilization |
| List of Open Files |
| PCB Pointer |

- A PCB is a data block or record containing many pieces (parts) of information associated with a specific process.
- Each process is represented in the operating system by its own control block (PCB).
- Each user process has a PCB which is created when a user creates a process and it is removed from the system when the process is terminated (killed).
- The PCB gives information about the status of the job (process).

1. <u>Process State</u>: The process state represents the current state of the process, which may be any of new, ready, running, waiting and terminated. The process may be in any of the above five states, and depending on the process state, PCB is updated.
2. <u>Program Counter</u>: Program counter indicates address of the next instruction to be executed. When a process goes to CPU burst (activity), the CPU (processor) has to know what instruction is to be executed. This is given by "<u>program counter</u>" in PCB.
3. <u>CPU Registers</u>: Contents of various registers such accumulator registers, index registers, general purpose registers are stored in PCB. When the process goes to I/O burst from CPU burst, it may so happen that the contents of these registers may be changed. When a process comes back to the CPU burst, the PCB contents must be restored.
4. <u>Memory Limits</u>: The information stored here is recording "Memory management", it depends on the memory system used by OS.

5. CPU Utilization: This information stores the details of CPU utilization such as in how m. time process (job) executed, and it contains process number.
6. List of Open Files: This information includes the list of I/O devices allocated to this process, and a list of open files.

......

## PROCESS SCHEDULING

CPU - I/O Burst Cycle:-
Process execution consists of a cycle of CPU execution (CPU Burst) and I/O wait. Process execution begins with a CPU burst that is followed by an I/O burst, which is followed by another CPU burst then another I/O burst, and so on. Finally the last CPU burst ends with a system request to terminate the execution.

CPU scheduler:-
The CPU scheduler selects a process from the processes (ready queue) in memory that are ready to execute and allocates the CPU to that process. The CPU scheduler also known as "SHORT-TERM SCHEDULER".

Context switch :-
When an interrupt occurs, the system needs to save the "current context" of the process which is currently running on the CPU. The context is represented in the PCB of the process. Switching the CPU to another process requires performing a "state save" of the current process and "state restore" of a different process. This task is known as a "CONTEXT SWITCH". When a context switch occurs, the kernel (OS) saves the context of the old process in its PCB, and loads the "saved context" of the new process.

Dispatcher:-
The dispatcher is a module (program) that gives control of the CPU to the process selected by the CPU scheduler. The dispatcher should be as fast as possible, since it is invoked during every process switch.

......

### Schedulers [*or*] Types of Schedulers

Scheduling is fundamental function of an Operating System. There are three types of schedulers:

1. LONG -TERM SCHEDULERS (LTS) or JOB SCHEDULER: LTS determines which jobs are admitted to the system for processing. It is a program that loads the selected jobs into main memory. These jobs are put into a "ready queue". Only a limited number of jobs (processes) allowed in the ready queue by LTS.
2. SHORT-TERM SCHEDULERS (STS) or CPU SCHEDULER: It is a program that is select one process/job from among the "ready processes" (ready queue), and allocating the CPU to that process. It decides which process is to be dispatched next for execution. The STS is called more frequently. For example: In UNIX OS, the STS is called once for every second.

3. MEDIUM TERM SCHEDULER (MTS): At one stage, the CPU utilization is maximum for specific number of user programs in memory. At this stage, if the degree of multiprogramming is further increased, CPU utilization drops. In this situation OS immediately calls MTS. The MTS will swap excess programs from memory and puts on disk. It means MTS performs "SWAP-OUT". After sometime, when some programs leave (terminate) memory, MTS will "SWAP-IN" those programs which were swapped out back into memory and execution starts. Thus SWAP-OUT and SWAP-IN should be done at appropriate time by MTS.

......

## PREEMPTIVE SCHEDULING V/S NON-PREEMPTIVE SCHEDULING

CPU scheduling decisions may take place under the following for circumstances:

1. When a process switches from the running state to the waiting state (for example: I/O request).
2. When a process switches from the running state to the ready state (for example: when an interrupt occurs).
3. When a process switches from the waiting state to the ready state (for example: completion of I/O).
4. When a process terminates.

When scheduling takes place only under the circumstances 1 & 4, we say the scheduling scheme is "Non-Preemptive" (Cooperative), otherwise it is "Preemptive".
Under Non-Preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method is used by "Microsoft Windows Operating System".

......

Imp-10M

## SCHEDULING ALGORITHMS [or] CPU SCHEDULING ALGORITHMS

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

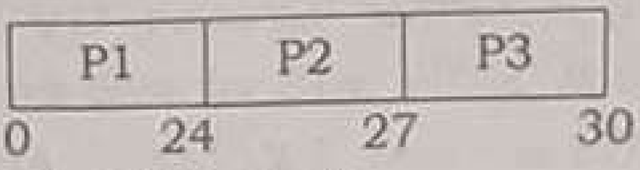
1. FIRST-COME FIRST-SERVED SCHEDULING: (FCFS)

FCFS is the simplest algorithm. In FCFS, the process that requests the CPU first is allocated to the CPU first. It is easily managed with FIFO (First In First Out) queue. When the CPU is free, the CPU is allocated to the process at the head of the queue. The running process is then removed from the queue. Consider the following set of processes along with the length of the CPU burst given in milliseconds (ms).

| Process | Burst Time (milliseconds) |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

If the processes arrive in the order P1, P2, and P3 we get the result shown in the following Gantt chart.

GANTT CHART:



Waiting time for process P1 = 0 milliseconds.
Waiting time for process P2 = 24 milliseconds.
Waiting time for process P3 = 27 milliseconds.

Average waiting time = $\dfrac{\text{process1 waiting time + process 2 waiting time + .......}}{\text{Total number of processes}}$

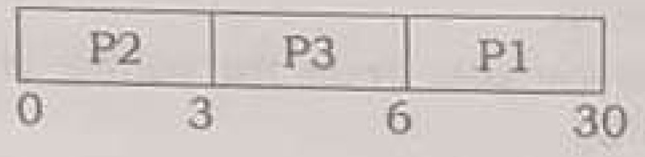
$= \dfrac{0+24+27}{3}$

$= 51/3$

= 17 milliseconds.

Suppose if the processes arrive in the order P2, P3, P1 then we get the following Gantt chart.

GANTT CHART:



Waiting time for process P1 = 6 ms
Waiting time for process P2 = 0 ms
Waiting time for process P3 = 3 ms

Average waiting time = $\dfrac{\text{process1 waiting time + process 2 waiting time +}}{\text{Total number of processes}}$ .......

$= \dfrac{6+0+3}{3}$

$= 9/3$

= 3 milliseconds.

The "***FCFS scheduling algorithm is non-preemptive***", it means 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.

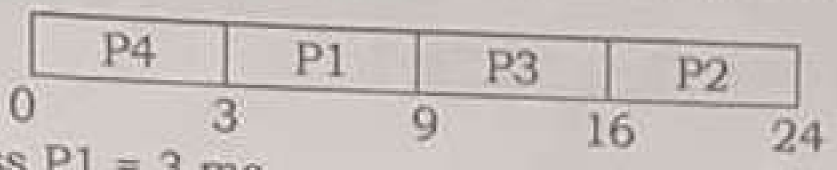
2. SHORTEST-JOB FIRST SCHEDULING: (SJF)

This algorithm schedules the processes by their CPU burst times. The process with the smallest CPU burst time will be processed before other processes. The CPU is assigned to the process that has the "smallest next CPU burst". Suppose if two processes have the same burst times then they will be scheduled through FCFS. Consider the following set of processes with the length of CPU burst given in milliseconds.

| Process | Burst Time (milliseconds) |
|---|---|
| P1 | 6 |
| P2 | 8 |
| P3 | 7 |
| P4 | 3 |

Using the SJF scheduling we would get the following Gantt chart.

GANTT CHART:



Waiting time for process P1 = 3 ms
Waiting time for process P2 = 16 ms
Waiting time for process P3 = 9 ms
Waiting time for process P4 = 0 ms

Average waiting time = $\frac{\text{process1 waiting time + process 2 waiting time + .......}}{\text{Total number of processes}}$

Average waiting time = $\frac{3+16+9+0}{4}$

= 28/4

= 7 ms.

This algorithm gives the minimum average waiting time for given set of processes. The SJF algorithm may be either ***preemptive*** or ***non-preemptive.*** When a new process arrives at the ready queue while the previous process is still executing. The new process may have a shorter CPU burst than what is left of the currently executing process, a "***Non-preemptive SJF algorithm***" will allow the currently running process to finish its CPU burst, whereas "***Preemptive SJF algorithm***" will preempt (prevent) the currently executive process.

## 3. PRIORITY SCHEDULING:

A priority is associated with each process. The CPU is allocated to the process with the highest priority. The equal priority processes are scheduled in FCFS order. Priorities are generally indicated by fixed range of numbers such as **0-7** or **0-4095**. Some systems use low number to represent high priority; others use low number for low priority.

Assume that here we use low number to represent high priority.

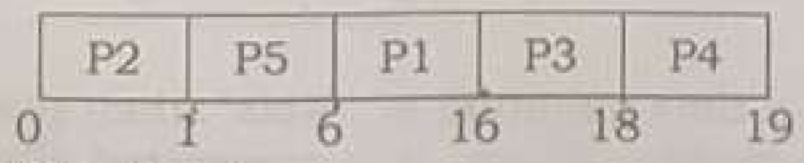
Consider the following set of processes arrived in the order P1, P2, P3, P4 and P5 with the length of CPU burst and associated priorities:

| Process | Burst Time | Priority |
|---|---|---|
| P1 | 10 | 3 |
| P2 | 1 | 1 |
| P3 | 2 | 3 |
| P4 | 1 | 4 |
| P5 | 5 | 2 |

$\longrightarrow$ HIGH PRIORITY

Using priority scheduling, the following Gantt chart is obtained.

GANTT CHART:



Waiting time of process P1 = 6 ms
Waiting time of process P2 = 0 ms
Waiting time of process P3 = 16 ms
Waiting time of process P4 = 18 ms
Waiting time of process P5 = 1 ms

the resultin

Average waiting time = $\frac{\text{process1 waiting time + process 2 waiting time + ......}}{\text{Total number of processes}}$

Average waiting time = $\frac{6+0+16+18+1}{5}$

= 41/5

= 8.2 ms.

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

## 4. ROUND-ROBIN SCHEDULING: [R-R]

The R-R scheduling algorithm is designed especially for "*Time Sharing Systems*". It is a similar to FCFS scheduling but preemption is added. A small unit of time called "***Time Quantum***" or "***Time Slice***" is defined. The ready queue is treated as a "***Circular queue***".

The new processes are added to the tail (end) of the ready queue. A Quantum of time is generally given in milliseconds. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after "***1Time Quantum***" and dispatches the process. Here one of two things will then happen:

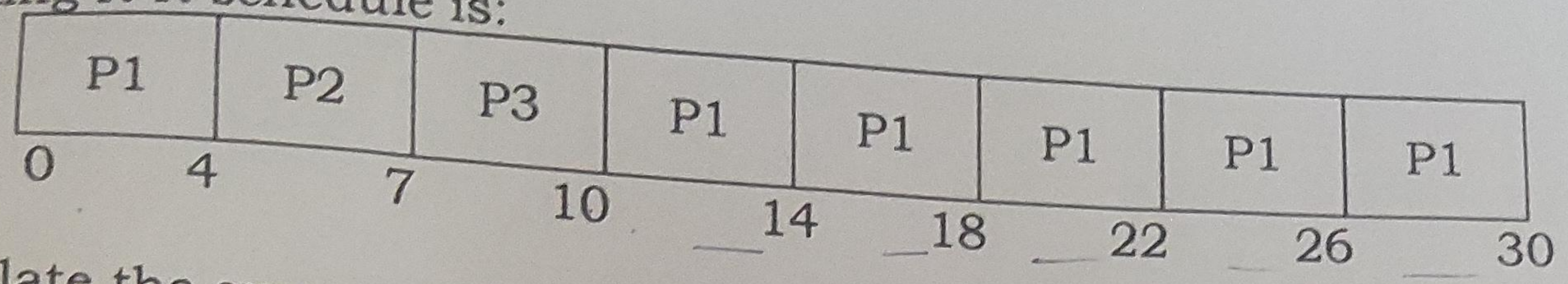
- The process may have a CPU burst of less than (<) 1time Quantum. In this case the process itself will release the CPU voluntarily. The scheduler will then proceeds to the next process in the ready queue.
- Otherwise if the CPU burst of the currently running processes is larger than (>) 1time Quantum,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

Consider the following set of processes that arrive at a time 0 (zero) with the length of the CPU burst given in milliseconds.

| Process | Burst Time (milliseconds) |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

Suppose we use a time quantum of "4ms", then process P1 gets the first for 4ms, since it requires another 20 ms, it is preempted after the first time quantum, and the CPU is given to the next process in the queue i.e. P2. Since P2 does not need 4ms, it quits before its time quantum expire. The CPU is given to the next process i.e. P3. Once each process has received the "1 Time Quantum", the CPU is returned to process P1 for an additional time quantum.

The resulting R-R schedule is:



Let's calculate the average waiting time for this schedule:

P1 waits for 6 (10-4) milliseconds
P2 waits for 4 milliseconds
P3 waits for 7 milliseconds.
The average waiting time= $\frac{6+4+7}{3}$
=17/3
= 5.6 ms.

......

## PREEMPTIVE SCHEDULING ALGORITHMS

1. SJF SCHEDULING ALGORITHM
2. PRIORITY SCHEDULING ALGORITHM
3. ROUND-ROBIN SCHEDULING ALGORITHM

## NON-PREEMPTIVE SCHEDULING ALGORITHMS

1. FCFS SCHEDULING ALGORITHM
2. SJF SCHEDULING ALGORITHM
3. PRIORITY SCHEDULING ALGORITHM

************