Process Concepts

Process

A process is a program in execution. For example, when we write a program in C or C++ and compile it, the compiler creates binary code. The original code and binary code are both programs. When we actually run the binary code, it becomes a process.

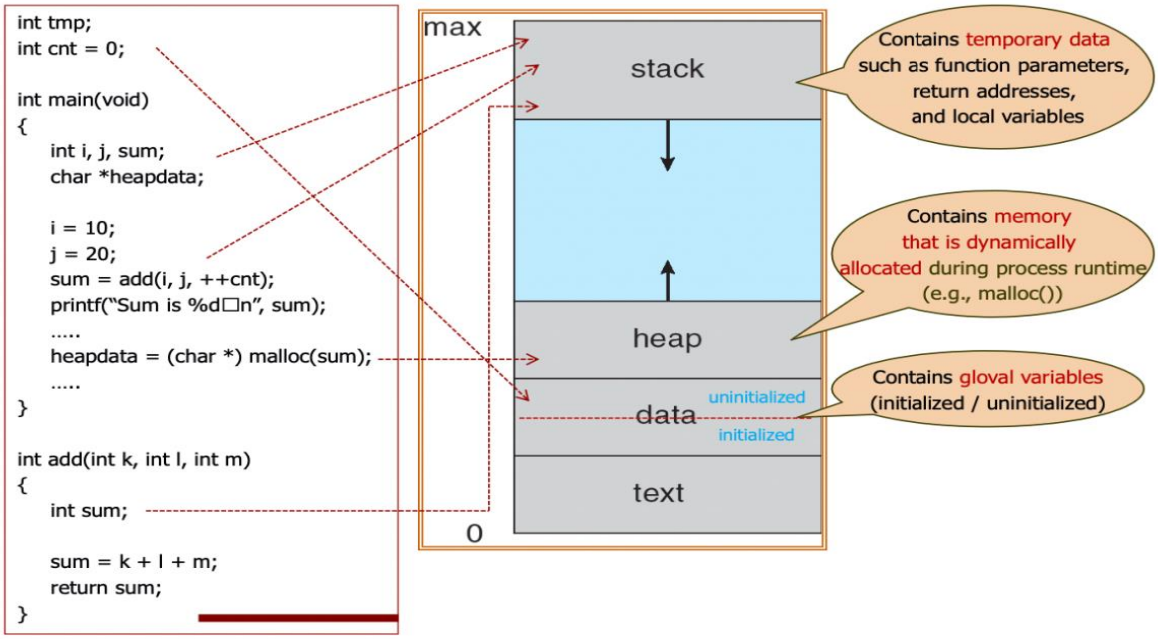
A process is an 'active' entity instead of a program, which is considered a 'passive' entity. A single program can create many processes when run multiple times; for example, when we open a .exe or binary file multiple times, multiple instances begin (multiple processes are created).

Process vs Program

- A process consists of instruction execution in machine code, but a program consists of instruction in any programming language.
- A process is a dynamic object, but a program is a static object.
- Process resides in main memory, but program resides in secondary memory.
- The period for the process is limited, but the period for the program is unlimited.
- A process is an active entity, and a program is a passive entity.

Process memory is divided into four sections for efficient working:

- The **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- The **Data section** is made up of the global and static variables, allocated and initialized prior to executing the main.
- The **Heap** is used for the dynamic memory allocation and is managed via calls to new, delete, malloc, free, etc.
- The **Stack** is used for local variables. Space on the stack is reserved for local variables when they are declared.



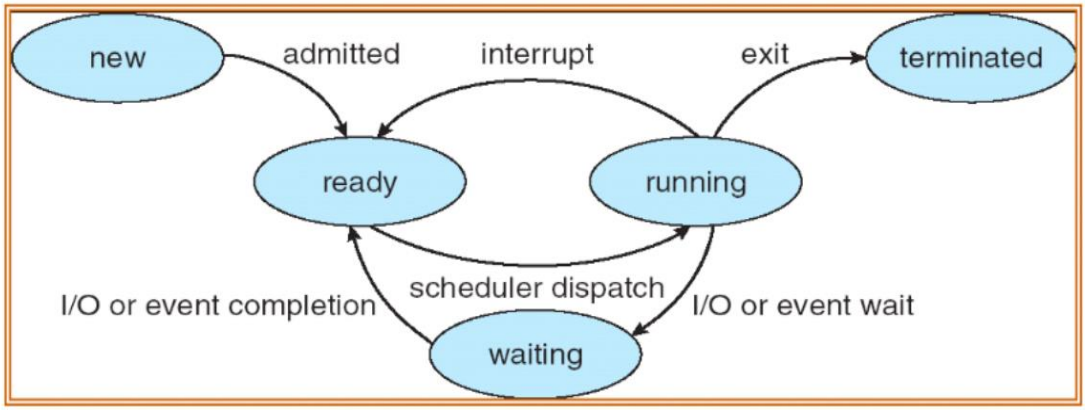
Process States

Process states in operating system are a way to manage resources efficiently by keeping track of the current state of each process.There are several process states in operating system, they are:

- **New State:** When a process is first created or is initialized by the operating system, it is in the new state. In this state, the process is being prepared to enter the ready state.
- **Ready State:** When a process is ready to execute, it is in the ready state. In this state, the process is waiting for the CPU to be allocated to it so that it can start executing its instructions. A process can remain in the ready state for an indeterminate period, waiting for the CPU to become available.
- **Running State:** When the CPU is allocated to a process, it enters the running state. In this state, the process executes its instructions and uses system resources such as memory, CPU, and I/O devices. Only one process can be in the running state at a time, and the operating system determines which process gets access to the CPU using scheduling algorithms.
- **Blocked State:** Sometimes, a process needs to wait for a particular event, such as user input or data from a disk drive. In such cases, the process enters the blocked state. In this state, the process is not using the CPU, but it is not ready to run either. The process remains in the blocked state until the event it is waiting for occurs.

- **Terminated State:** The terminated state is reached when a process completes its execution or terminates by the operating system. In this state, the process no longer uses any system resources, and its memory space is deallocated.

Now let us see the state diagram of these process states −



Step 1 − Whenever a new process is created, it is admitted into ready state.

Step 2 − If no other process is present at running state, it is dispatched to running based on scheduler dispatcher.

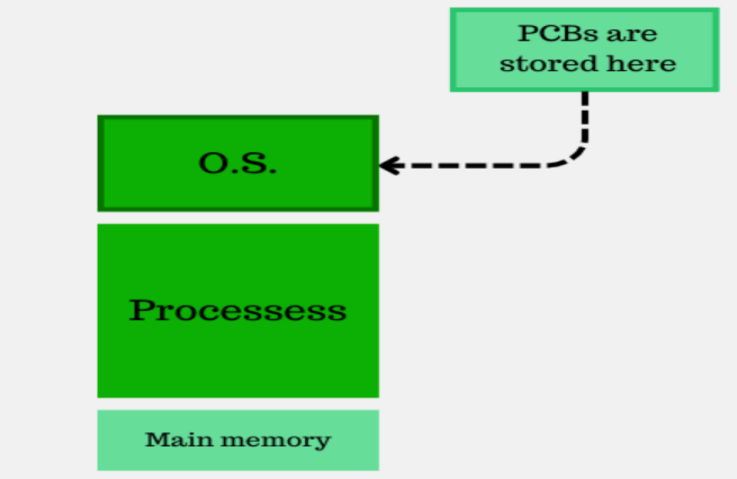
Step 3 − If any higher priority process is ready, the uncompleted process will be sent to the waiting state from the running state.

Step 4 − Whenever I/O or event is completed the process will send back to ready state based on the **interrupt signal** given by the running state.

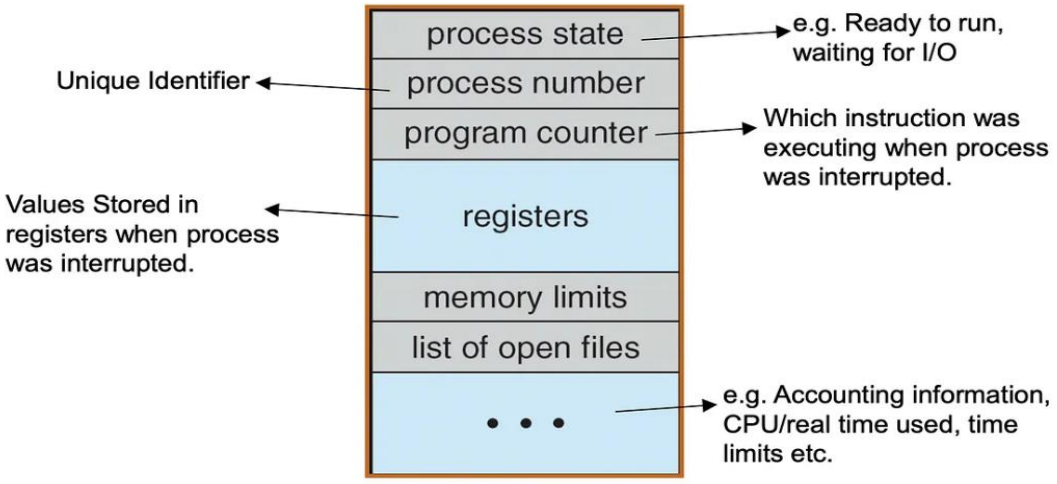
Step 5 − Whenever the execution of a process is completed in running state, it will exit to terminate state, which is the completion of process.

Process Control Block

Process Control Block is a data structure that contains information of the process related to it. The process control block is also known as a task control block, entry of the process table, etc. Each time a process is created, a unique PCB is created and is removed when the process is completed. The PCB is created with the aim of helping the OS to manage the enormous amounts of tasks that are being carried out in the system.



PCB Contains the following attributes



Process State

Throughout its existence, each process goes through various phases. The present state of the process is defined by the process state.

Process Number

When a new process is created by the user, the operating system assigns a unique ID i.e. a process-ID to that process. This ID helps the process to be distinguished from other processes existing in the system.The operating system has a limit on the maximum number of processes it is capable of dealing with, let's say the OS can handle most N processes at a time.

So, process-ID will get the values from 0 to N-1.

Program Counter

The address of the next instruction to be executed is specified by the program counter. The address of the program's first instruction is used to initialize the program counter before it is executed.

The value of the program counter is incremented automatically to refer to the next instruction when each instruction is executed. This process continues till the program ends.

Registers

The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

List of open files

This contains information about those files that are used by that process. This helps the operating system close all the opened files at the termination state of the process.

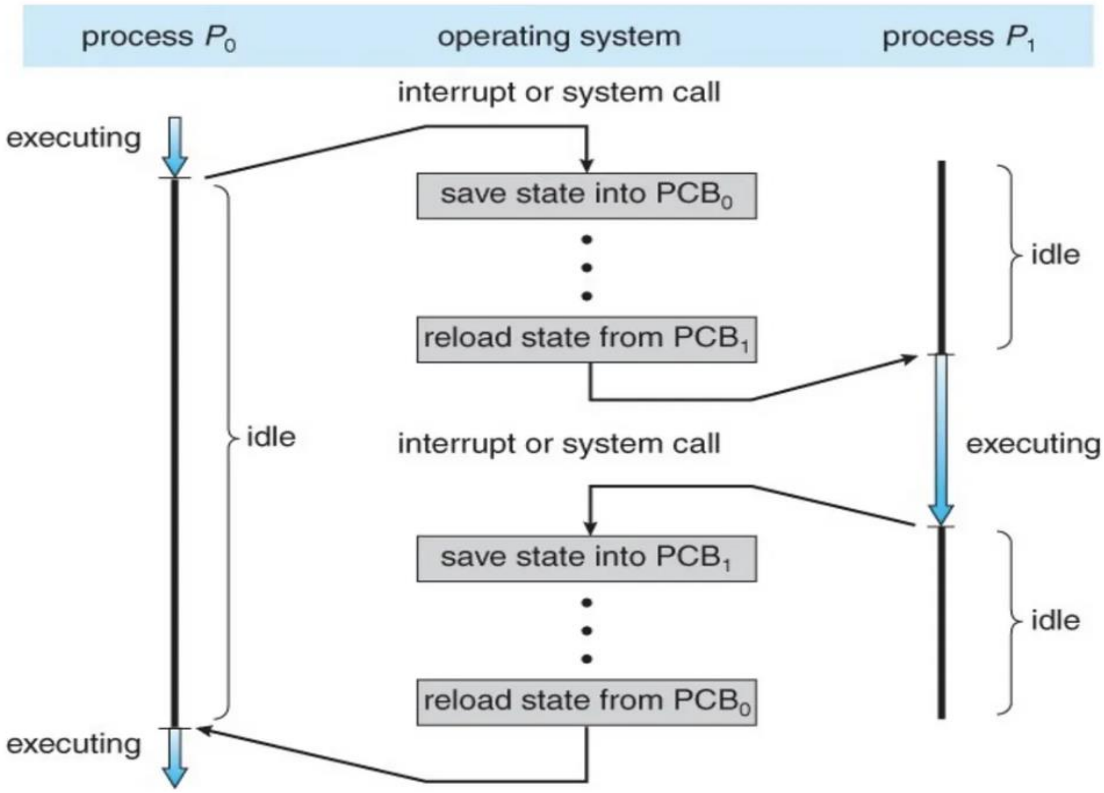
Context Switching

A context switching is a mechanism that involves switching of the CPU from one process or task to another. While peforming switching the operating system saves the state of a running process so it can be resumed later, and then loads the state of another process. A context switching allows a single CPU to handle multiple process requests simultaneously without the need for any additional processors.

Following are the reasons that describe the need for context switching in the Operating system.

1. If a high priority process falls into the ready queue, the currently running process will be shut down or stopped by a high priority process to complete its tasks in the system.
2. If any running process requires I/O resources in the system, the current process will be switched by another process to use the CPUs. And when the I/O requirement is met, the old process goes into a ready state to wait for its execution in the CPU. Context switching stores the state of the process to resume its tasks in an operating system. Otherwise, the process needs to restart its execution from the initials level.
3. If any interrupts occur while running a process in the operating system, the process status is saved as registers using context switching. After resolving the interrupts, the process switches from a wait state to a ready state to resume its execution at the same point later, where the operating system interrupted occurs.

.



The following steps describe the basic sequence of events when moving between processes:

1. The CPU executes Process 0.
2. A triggering event occurs, such as an interrupt or system call.
3. The system pauses Process 0 and saves its state (context) to PCB 0, the process control block that was created for that process.
4. The system selects Process 1 from the queue and loads the process's state from PCB 1.
5. The CPU executes Process 1, picking up from where it left off (if the process had already been running).
6. When the next triggering event occurs, the system pauses Process 1 and saves its state to PCB 1.
7. The Process 0 state is reloaded, and the CPU executes the process, once again picking up from where it left off. Process 1 remains in an idle state until it is called again.

Advantage of Context Switching

Context switching is used to achieve multitasking . Multitasking gives an illusion to the users that more than one process are being executed at the same time. But in reality, only one task is being executed at a particular instant of time by a processor. Here, the context

switching is so fast that the user feels that the CPU is executing more than one task at the same time.

The disadvantage of Context Switching

The disadvantage of context switching is that it requires some time for context switching i.e. the context switching time. Time is required to save the context of one process that is in the running state and then getting the context of another process that is about to come in the running state. During that time, there is no useful work done by the CPU from the user perspective. So, context switching is pure overhead in this condition.

Process Operations

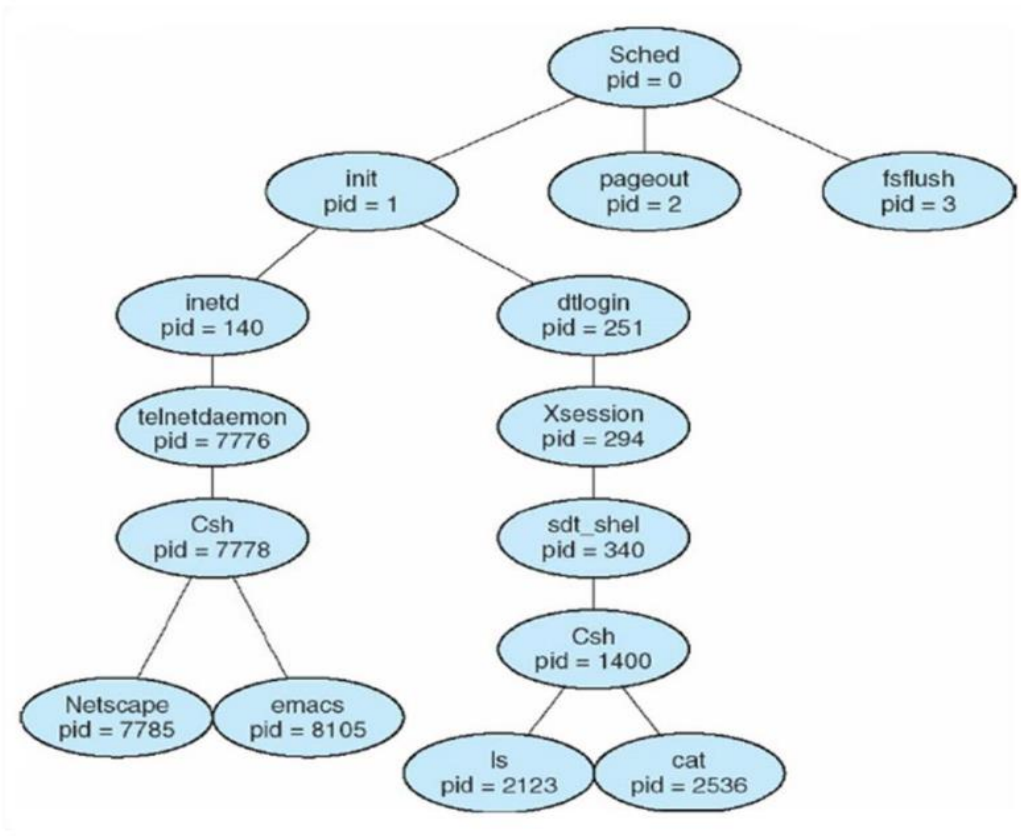
The operations of process carried out by an operating system are primarily of two types:

1. Process creation
2. Process termination

Process creation

Process creation in operating systems (OS) is a fundamental operation that involves the initiation of a new process. The creating process is called the **parent** while the created process is called the **child.** Each child process may in turn create new child process. Every process in the system is identified with a process identifier**(PID)** which is unique for each process. A process uses a System call to create a child process.

Figure Below illustrates a typical process tree for the Solaris operating system, showing the name of each process (daemon) and its pid.



In Linux and other Unix-like operating systems, a daemon is a background process that runs independently of any interactive user session. Daemons typically start at system boot and run continuously until the system is shut down. They perform various system-level tasks, such as

managing hardware devices, handling network requests, scheduling jobs, or any other kind of service that needs to run in the background without direct user intervention

Daemon	Operation
init	The **init** process is the first process started by the Linux/Unix kernel and holds the process ID (PID) 1.
pageout	The **pageout** daemon is responsible for managing virtual memory,
fsflush	The **fsflush** process is a background daemon responsible for synchronizing cached data to disk.
inetd	**inetd**, short for "internet super-server", is a daemon that listens on designated for incoming connections and then launches the appropriate program. inetd is responsible for networking services such as telnet and ftp
dtlogin	**Desktop login- dtlogin** is the display manager component responsible for managing user logins to the graphical environment.

In Solaris, the process at the top of the tree is the sched process, with pid of 0. The sched process creates several children processes-including pageout and fsflush. The sched process also creates the init process, which serves as the root parent process for all user processes. we see two children of init- inetd and dtlogin. When a user logs in, dtlogin creates an X-windows session (Xsession), which in turns creates the sdt_shel process. (**sdt_shel**, part of the CDE (Common Desktop Environment), is a graphical user interface tool designed to provide users with an easy way to access various system functions and applications). Below sdt_shel, a user's command-line shell-the C-shell or csh-is created. In this command line interface, the user can then invoke various child processes, such as the ls and cat commands. We also see a csh process with pid of 7778 representing a user who has logged onto the system using telnet. This user has started the Netscape browser (pid of 7785) and the emacs editor (pid of 8105). (Emacs is a text editor which provides a robust platform for executing complex editing tasks, writing code in various programming languages, managing files, reading emails, browsing the web, and even playing games.)

Resource Sharing Between the process

- In general, a process will need certain resources (CPU time, memory, files, I/0 devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.

- The parent may have to partition its resources among its children, or it may be able to share some resources (such as Memory or files) among several of its children.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.
- In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process.
- For example, consider a process whose function is to display the contents of a file-say, img.jpg-on the screen of a terminal. When it is created, it will get, as an input from its parent process, the name of the file img.jpg, and it will use that file name, open the file, and write the contents out.
- It may also get the name of the output device. Some operating systems pass resources to child processes. On such a system, the new process may get two open files, img.jpg and the terminal device, and may simply transfer the datum between the two.

Process Execution

When a process creates a new process, two possibilities exist in terms of execution:

- The parent continues to execute concurrently with its children.
- The parent waits until some or all of its children have terminated.

Process AdressSpace

There are also two possibilities in terms of the address space of the new process:

- The child process is a duplicate of the parent process (it has the same program and data as the parent). (Ex:fork() System Call)
- The child process has a new program loaded into it.(Ex: exec() System call)

Process termination

➢ A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process-including physical and virtual memory, open files, and I/0 buffers-are deallocated by the operating system.

➢ Termination can occur in other circumstances as well A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess () in Win32). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs. Note that a parent needs to know the identities of its children.

Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

➢ A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. if a Parent process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination.**

System calls - Process Creation & termination in Linux

1.fork() System call

In Linux OS a new process is created by the fork () system call.

- The "fork()" system call is used to create a new process, known as the child process, from an existing process, known as the parent process. When "fork()" is called, the operating system creates a new process by duplicating the entire address space (memory), file descriptors, and other attributes of the parent process.
- The child process is an exact copy of the parent process at the time of the "fork()" call. However, the child process has its own process ID (PID) and is assigned a unique PID by the operating system. Both the parent and child processes then continue execution from the point immediately after the "fork()" call.
- The "fork()" system call returns different values to the parent and child processes. In the parent process, the return value is the PID of the child process, while in the child process, the return value is 0. This allows the parent and child processes to differentiate themselves and execute different sections of code if needed.
- **Return value of fork()** On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

The fork() system call is often used to implement process spawning, where a parent process creates multiple child processes to perform tasks concurrently or in a parallel manner.

2.exec() System Call

- The “exec()” system call is used to replace the current process with a new program or executable. It loads a new program into the current process’s memory, overwriting the existing program’s code, data, and stack.
- When “exec()” is called, the operating system performs the following steps:
  a. The current process’s memory is cleared, removing the old program’s instructions and data.
  b. The new program is loaded into memory.
  c. The program’s entry point is set as the starting point of execution.
  d. The new program’s code begins execution.
- The “exec()” system call is often used after a “fork()” call in order to replace the child process’s code with a different program. This allows the child process to execute a different program while preserving the parent process’s execution.
  Together, the “fork()” and “exec()” system calls enable processes to create child processes and replace their own execution with different programs, facilitating process creation, concurrency, and program execution in operating systems.

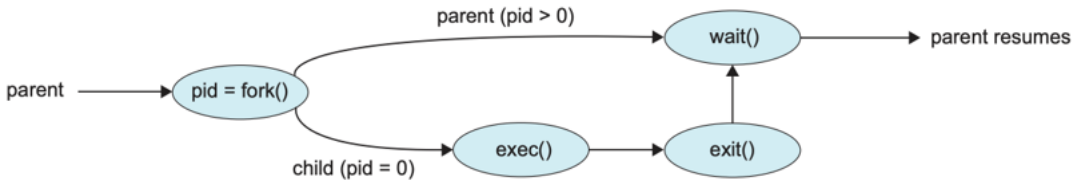
3.Wait() System call

- In some systems, a process may have to wait for another process to complete its execution before proceeding. When a parent process makes a child process, the parent process execution is suspended until the child process is finished. The **wait()** system call is used to suspend the parent process. Once the child process has completed its execution, control is returned to the parent process.
- wait() system call takes only one parameter which stores the status information of the process. Pass NULL as the value if you do not want to know the exit status of the child process and are simply concerned with making the parent wait for the child.
- On success, wait returns the PID of the terminated child process while on failure it returns -1. (pid_t wait(int *wstatus))

4.exit() System call

The exit() is a system call that is used to terminate the process. After the use of exit() system call, all the resources used in the process are retrieved by the operating system and then terminate the process.

Lets depict all the system calls in the form of a process transition diagram.



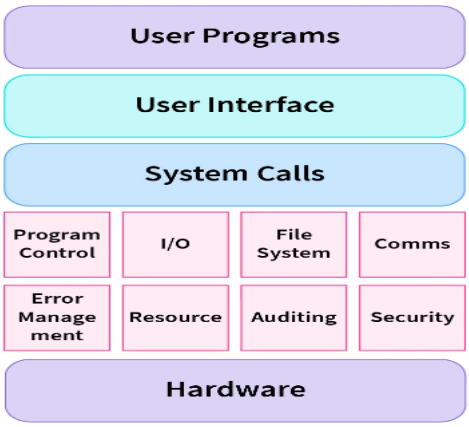
A process begins its life when it is created. A process goes through different states before it gets terminated.

- The first state that any process goes through is the creation of itself. Process creation happens through the use of fork() system call, (*A system call is a function that a user program uses to ask the operating system for a particular service .System calls serve as the interface between an operating system and a process*) which creates a new process(child process) by duplicating an existing one(parent process). The process that calls fork() is the parent, whereas the new process is the child.
- In most cases, we may want to execute a different program in child process than the parent. The exec() family of function calls creates a new address space and loads a new program into it.
- Finally, a process exits or terminates using the exit() system call. This function frees all the resources held by the process(except for pcb).
- A parent process can enquire about the status of a terminated child using wait() system call. When the parent process uses wait() system call, the parent process is blocked till the child on which it is waiting terminates.

UNIX SYSTEM CALLS	DESCRIPTION	WINDOWS API CALLS	DESCRIPTION
Process Control			
`fork()`	Create a new process.	`CreateProcess()`	Create a new process.
`exit()`	Terminate the current process.	`ExitProcess()`	Terminate the current process.
`wait()`	Make a process wait until its child processes terminate.	`WaitForSingleObject()`	Wait for a process or thread to terminate.
`exec()`	Execute a new program in a process.	`CreateProcess()` or `ShellExecute()`	Execute a new program in a new process.
`getpid()`	Get the unique process ID.	`GetCurrentProcessId()`	Get the unique process ID.

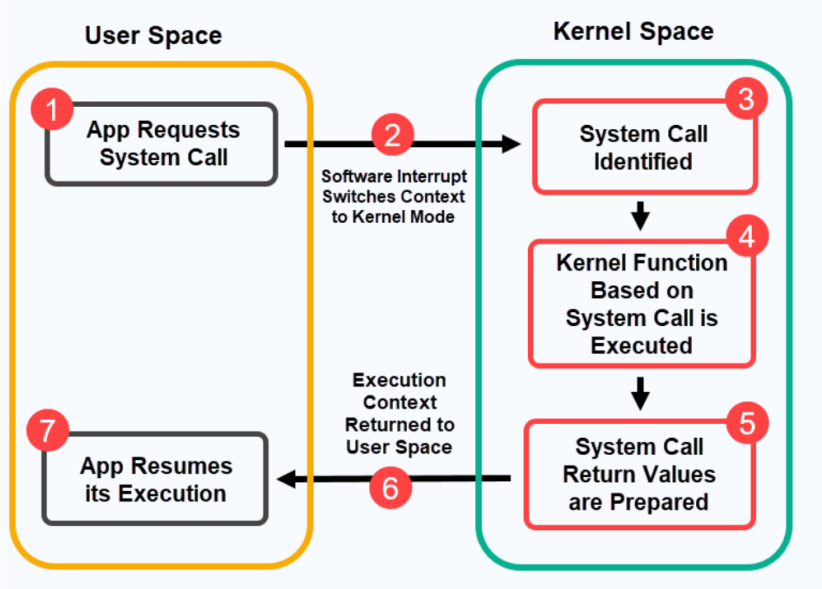
System call interface for process management

A system call is a function that a user program uses to ask the operating system for a particular service which cannot be carried out by normal function. System calls provide the interface between the process and the operating system.



Working of System calls

This high-level overview explains how system calls work:



1. **System Call Request**. The application requests a system call by invoking its corresponding function. For instance, the program might use the **read()** function to read data from a file.
2. **Context Switch to Kernel Space**. A software interrupt or special instruction is used to trigger a context switch and transition from the user mode to the kernel mode.
3. **System Call Identified**. The system uses an index to identify the system call and address the corresponding kernel function.

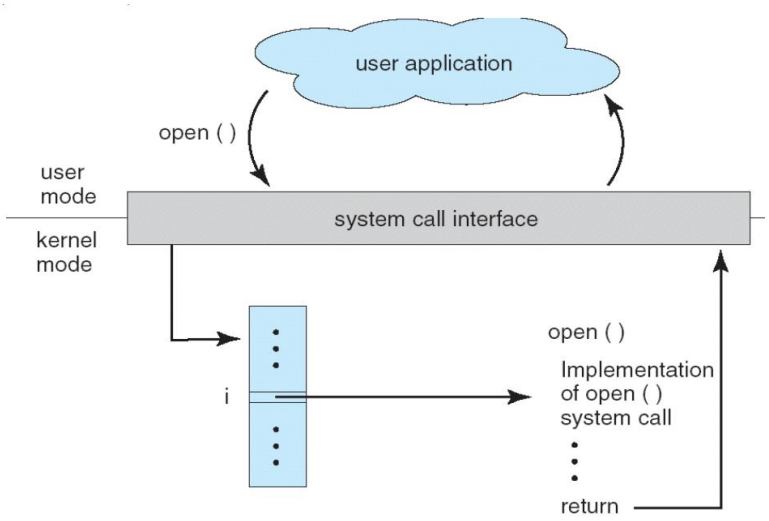
4. **Kernel Function Executed**. The kernel function corresponding to the system call is executed. For example, reading data from a file.

5.**System Prepares Return Values**. After the kernel function completes its operation, any return values or results are prepared for the user application.

6. **Context Switch to User Space**. The execution context is switched back from kernel mode to user mode.

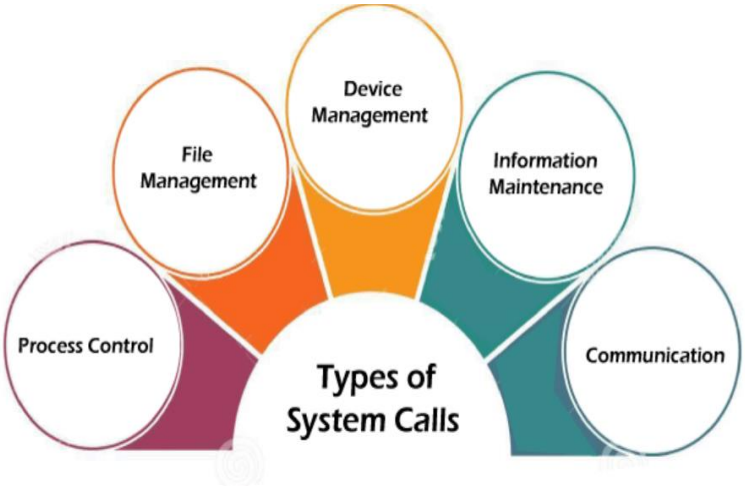
7. **Resume Application**. The application resumes its execution from where it left off, now with the results or effects of the system call.

Example



System calls are divided into 5 categories mainly :

- Process Control
- File Management
- Device Management
- Information Maintenance
- Communication



Process Management System Calls

System calls in Linux	Description
fork	Create a new process.
vfork	Replace the current process with a new one.
exit	Terminate a process
wait	It makes a parent process stop its execution till the termination of the child process.
waitpid	It makes a parent process stop its execution till the termination of the specified child process.(Multiple child process)
exec	It loads a new program in child process

fork()

It is the primary method of process creation on Unix-like operatingsystems. This function creates a new copy called the child out of the original process, that is called the parent. When the parent process closes or crashes for some reason, it also kills the child process.

Syntax:

pid=fork();

- The operating system is using a unique id for every process to keep track of all processes. And for that, fork() doesn't take any parameter and return an int value as following:
- **Zero**: if it is the child process (the process created). **Positive value**: if it is the parent process.
- The difference is that, in the parent process, fork() returns a value which represents the **process ID** of the child process. But in the child process, fork() returns the value 0.
- **Negative value**: if an error occurred.

Sample Program

/*When working with fork(), <sys/types.h> can be used for type ***pid_t*** for processes ID's as pid_t is defined in <sys/types.h>.

The header file <unistd.h> is where fork() is defined so you have to include it to your program to use fork().*/

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   /* fork a process */
   pid_t p;
   p = fork();
   /* the child and parent will execute every line of code after the fork (each separately) */
   if(p == 0)
      printf("Hello child pid=%d\n", getpid());
   else if(p > 0)
      printf("Hello parent pid=%d\n", getpid());
   else
      printf("Error: fork() failed\n");
   return 0;
}
```

The output will be:

```
Hello parent pid=1601
Hello child pid=1602
```

vfork()

The ***vfork()*** is another system call that is utilized to make a new process. The child process is the new process formed by the ***vfork()*** system call, while the parent process is the process that uses the ***vfork()*** system call. The vfork() system call doesn't create separate address spaces for both parent and child processes.

- Because the child and parent processes share the same address space,the child process halts the parent process's execution until the child process completes its execution.
- . If one of the processes changes the code, it is visual to the other process transferring the same pages. Assume that the parent process modifies the code; it will be reflected in the child process code.

fork()	**vfork()**
Child process and parent process has separate address spaces.	Child process and parent process shares the same address space.
Parent and child process execute simultaneously.	Parent process remains suspended till child process completes its execution.
If the child process alters any page in the address space, it is invisible to the parent process as the address space are separate.	If child process alters any page in the address space, it is visible to the parent process as they share the same address space.

Syntax:

pid=vfork();

Sample Program

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   /* fork a process */
   pid_t p;
   p = vfork();
   /* the child and parent will execute every line of code after the fork (each separately) */
   if(p == 0)
{
// Child process
```

```
        printf("This is the child process. PID: %d\n", getpid());
        printf("Child process is exiting with exit() \n");
        exit(0);
}
else if(p > 0)
{
// Parent process might wait for child process to terminate
        printf("This is the parent process. PID: %d\n", getpid());
}
        else
        printf("Error: fork() failed\n");
    return 0;
}
```

output

This is the child process. PID: 91

Child process is exiting with exit()

This is the parent process. PID: 90

Wait()

This system call is used in processes that have a parent-child relationship. It makes a parent process stop its execution till the termination of the child process. The program creates a child process via the fork() system call and then calls the wait() system call to wait for the child process to finish its execution.

Syntax

Below, we can see the syntax for the wait() system call. To use this in our C programs, we will have to include the Standard C library sys/wait.h.

```
pid_t wait(int *status);
```

Parameters and return value

The system call takes one argument named status. This argument represents a pointer to an integer that will store the exit status of the terminated child program.we can even replace status with NULL parameter.

When the system call completes, it can return the following.

- **Process ID:** The process ID of the child process that is terminated, which has the object type pid_t.

- **Error value:** If there is any error during system call execution, it will return -1, which can be used for error handling.

Sample program

```
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
pid-t  p,childpid;
   p = fork();
   // code for the child process
   if(p== 0){
      printf("Child: I am running!!\n\n");
      printf("Child: I have PID: %d\n\n", getpid());
      exit();
   }
 // code for the parent process
   else{
      // print parent running message
      printf("Parent: I am running and waiting for child to finish!!\n\n");
      // call wait system call
      childpid = wait(NULL);
      // print the details of the child process
      printf("Parent: Child finished execution!, It had the PID: %d,\n", childpid);
   }
   return 0;
}
```

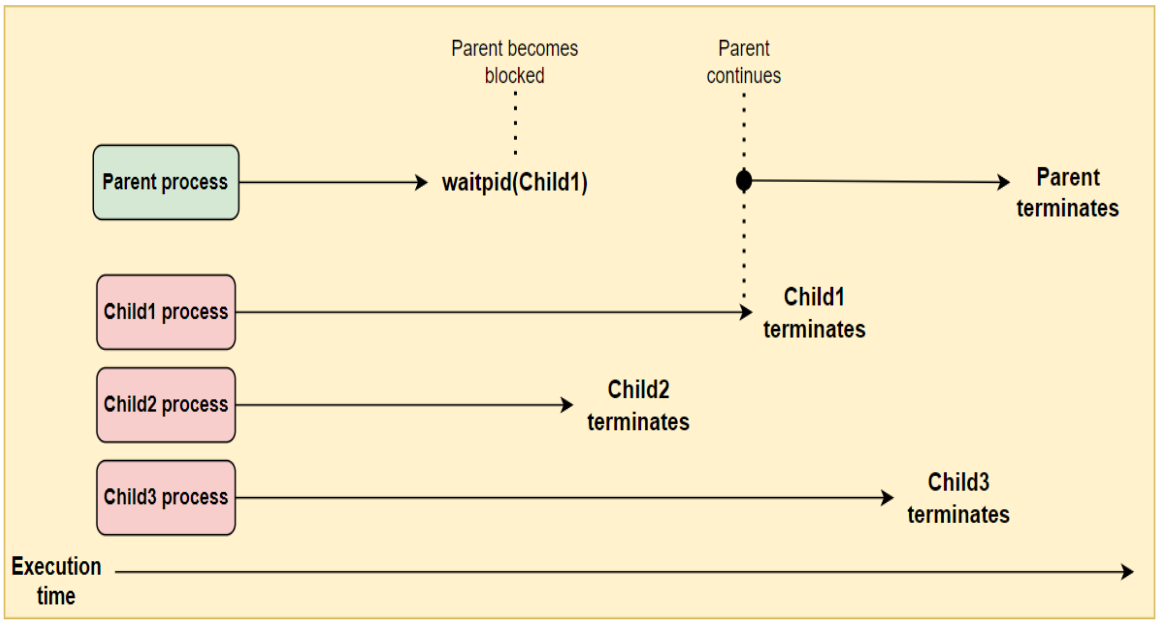
Output:

```
Parent: I am running and waiting for child to finish!!
Child: I am running!!
Child: I have PID: 102
Parent: Child finished execution!, It had the PID: 102
```

waitpid()

This system call waits for a specific process to finish its execution. This system call can be accessed using our C programs' library sys/wait.h.

Let's understand how the waitpid() function works with the help of a diagram.



The diagram shows that we have a parent process with 3 child processes, namely, Child1, Child2, and Child3. If we want the parent to wait for a specific child, we could use the waitpid() function.

In the diagram above, we made the parent process wait for Child1 to finish its execution using the function. When the parent process is called, the waitpid() function gets blocked by the operating system and can't continue its execution till the specified child process Child1 is terminated.

If we were to replace the process in the waitpid() function with Child3, then the parent would only continue when Child3 had terminated.

Syntax

pid_t waitpid(pid_t pid, int *status_ptr, int options);

Let's discuss the three arguments that we provide to the system call.

- pid: Here, we provide the process ID of the process we want to wait for. If the provided pid is 0, it will wait for any arbitrary child to finish.
- status_ptr: This is an integer pointer used to access the child's exit value. If we want to ignore the exit value, we can use NULL here.
- options: Here, we can add additional flags to modify the function's behavior. The various flags are discussed below:
    - WCONTINUED: It is used to report the status of any child process that has been terminated and those that have resumed their execution after being stopped.

- WNOHANG: It is used when we want to retrieve the status information immediately when the system call is executed. If the status information is not available, it returns an error.
- WUNTRACED: It is used to report any child process that has stopped or terminated.

Return value

The system call will return the process ID of the child process that was terminated. If there is any error while waiting for the child process via the waitpid() system call, it will return -1, which corresponds to an error.

Sample Program

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
   pid_t pids[2], wpid;
   // Fork first child process
   pids[0] = fork();
   if (pids[0] == 0) {
      // First child process
      printf("First child process: PID = %d\n", getpid());
     printf("First child process exiting\n");
     exit(0); // First child exits
   }
   // Fork second child process
   pids[1] = fork();
   if (pids[1] == 0)
{
      // Second child process
      printf("Second child process: PID = %d\n", getpid());
      printf("Second child process exiting\n");
     exit(0); // Second child exits
   }

if(pid[0]>0 && pid[1]>0)
```

```
{
 // Parent process: wait specifically for the first child
    wpid = waitpid(pids[1], NULL,0);
        printf("Parent: Proceeding after the second child with pid =%d has finished.\n",pid[1]);
}
    return 0;
}
```

Output

First child process: PID = 119

First child process exiting

Second child process: PID = 120

Second child process exiting

Parent: Proceeding after the second child with pid =120 has finished

exec()

The "exec()" system call is used to replace the current process with a new program or executable. It loads a new program into the current process's memory, overwriting the existing program's code, data, and stack.

When "exec()" is called, the operating system performs the following steps:

a. The current process's memory is cleared, removing the old program's instructions and data.

b. The new program is loaded into memory.

c. The program's entry point is set as the starting point of execution.

d. The new program's code begins execution.

The "exec()" system call is often used after a "fork()" call in order to replace the child process's code with a different program. This allows the child process to execute a different program while preserving the parent process's execution.

Exec system call is a collection of functions and in C programming language, the standard names for these functions are as follows:

(l- represents list, v represents vector)

execl(): Executes a program specified by a pathname, and arguments are passed as a variable-length list of strings terminated by a NULL pointer.

Syntax: int execl(const char *path, const char *arg, ..., NULL);

***Parameters**:*

- **path**: Specifies the path to the executable file which the process will run.
- **arg**: Represents the list of arguments to be passed to the executable. The first argument typically is the name of the program being executed. The list of arguments must be terminated by **NULL** to indicate the end of the argument list.

***Return Value**:*

- On success, **execl()** does not return; the new program image takes over the process.
- If an error occurs, it returns **-1**, and **errno** is set to indicate the error.

execv(): Executes a program specified by a pathname, and arguments are passed as an array of strings terminated by a NULL pointer.

Syntax: int execv(const char *path, char *const argv[]);

***Parameters**:*

- **path**: Specifies the path to the executable file which the process will run.
- **argv**: An array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should be the name of the executed program. The array of pointers must be terminated by a NULL pointer.

***Return Value**:*

- On success, **execv()** does not return because the current process image is replaced by the new program image.
- On failure, it returns **-1**, and **errno** is set to indicate the error.

execlp(): Similar to **execl()**, but the program is searched for in the directories specified by the PATH environment variable.

Syntax : int execlp(const char *file, const char *arg, ..., NULL);

***Parameters**:*

- **file**: The name of the executable file to run. If this name contains a slash (/), then **execlp()** will treat it as a path and will not search the **PATH** environment variable.
- **arg**: Represents the list of arguments to be passed to the executable. The first argument, by convention, should be the name of the program being executed. The list of arguments must be terminated by **NULL** to indicate the end of the argument list.

***Return Value**:*

- On success, **execlp()** does not return; the new program image takes over the process.
- On failure, it returns **-1**, and **errno** is set to indicate the error.

execvp(): Similar to **execv()**, but the program is searched for in the directories specified by the PATH environment variable.

Syntax: int execvp(const char *file, char *const argv[]);

Parameters:

- **file**: The name of the executable file to run. If this name contains a slash (/), then **execvp()** will treat it as a path and will not search the **PATH** environment variable.
- **argv**: An array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should be the name of the executed program. The array of pointers must be terminated by a NULL pointer to indicate the end of the argument list.

Return Value*:*

- On success, **execvp()** does not return because the current process image is replaced by the new program image.
- On failure, it returns **-1**, and **errno** is set to indicate the error.

execle(): Similar to **execl()**, but allows specifying the environment of the executed program as an array of strings.

Syntax*:* int execle(const char *path, const char *arg, ..., char *const envp[]);

Parameters*:*

- **path**: Specifies the path to the executable file which the process will run.
- **arg**: Represents the list of arguments to be passed to the executable. The first argument, by convention, should be the name of the program being executed. The list of arguments must be terminated by **NULL** to indicate the end of the argument list.
- **...**: A variable number of arguments representing the arguments to be passed to the executable, terminated by a **NULL** pointer.
- **envp[]**: An array of strings, conventionally of the form **key=value**, which are passed as the environment of the new program. The array must be terminated by a **NULL** pointer.

Return Value*:*

- On success, **execle()** does not return because the current process image is replaced by the new program image.
- On failure, it returns **-1**, and **errno** is set to indicate the error.

execve(): The most general form, which executes a program specified by a pathname, takes an array of arguments and an array of environment variables.

Syntax: int execve(const char *pathname, char *const argv[], char *const envp[]);

Parameters:

- **pathname**: Specifies the file path of the executable file which the process will run. This file must be a binary executable or a script with a shebang (**#!**) line.
- **argv[]**: An array of pointers to null-terminated strings that represent the argument list to be passed to the new program. The first argument, by convention, should be the name of the executed program. The array of pointers must be terminated by a NULL pointer.
- **envp[]**: An array of strings, conventionally of the form **key=value**, which are passed as the environment of the new program. The array must be terminated by a NULL pointer.

Return Value:

- On success, **execve()** does not return because the current process image is replaced by the new program image.
- On failure, it returns **-1**, and **errno** is set to indicate the error.

Sample Program

```
include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
   pid_t pid = fork();
 if (pid == 0) {
       // Child process
       printf("Child process (PID %d): executing '/bin/ls'\n", getpid());
       // Replace the child process with '/bin/ls'
       // execl() takes the path to the program, the name of the program,
       // and any command-line arguments, ending with a NULL pointer.
       execl("/bin/ls", "ls", NULL);
  }
else
{       // Parent process
       printf("Parent process (PID %d): waiting for child process\n", getpid());
```

```
    }
  }
```

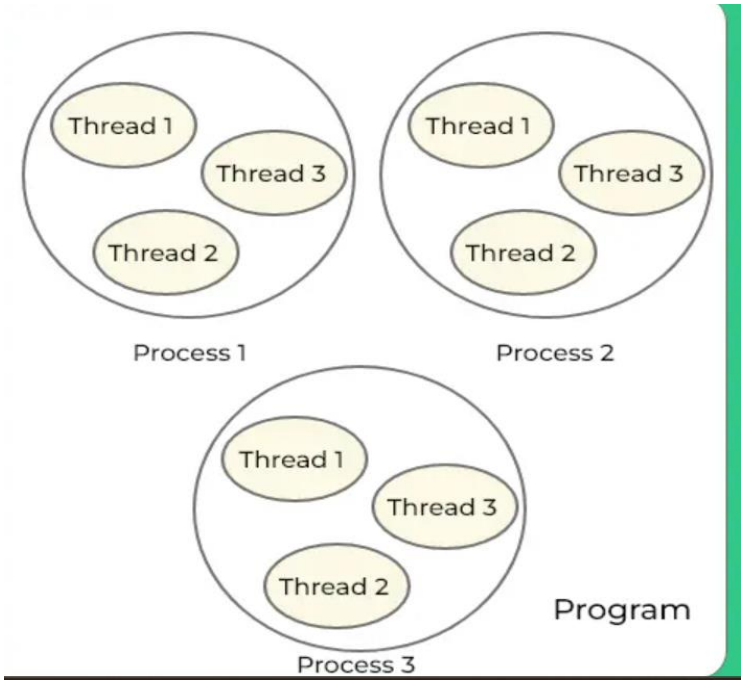
output

```
Parent process (PID 142): waiting for child process
Child process (PID 143): executing '/bin/ls'
 a.out  exec.c  fork.c  vfork.c  wait.c  waitpid  waitpid.c
```

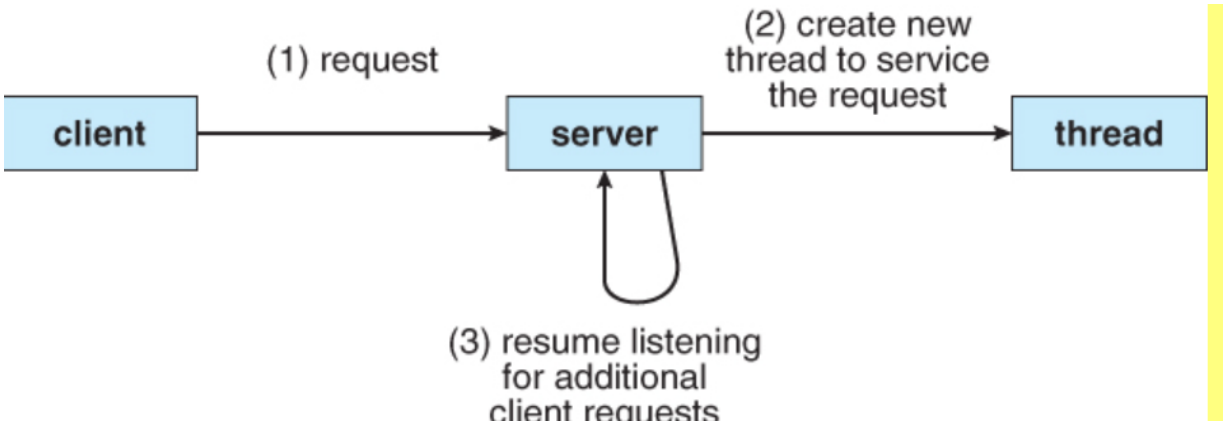
Introduction to Threads

Program, **Process** and **Threads** are three basic concepts of the operating systems.

- Program is an executable file containing the set of instructions written to perform a specific job on your computer. Programs are not stored on the primary memory in your computer. They are stored on a disk or a secondary memory on your computer. They are read into the primary memory and executed by the kernel. A program is sometimes referred as **passive entity** as it resides on a secondary memory.
- Process is an executing instance of a program. A process is sometimes referred as **active entity** as it resides on the primary memory and leaves the memory if the system is rebooted. Several
- Thread is the smallest executable unit of a process. A thread is often referred to as a lightweight process due to its ability to run sequences of instructions independently while sharing the same memory space and resources of a process. A process can have multiple threads. Each thread will have their own task and own path of execution in a process. Threads are popularly used to improve the application through **parallelism** . Actually only one thread is executed at a time by the CPU, but the **CPU switches rapidly** between the threads to give an illusion that the threads are running parallelly.



- ***For example, in a browser, multiple tabs can be different threads or While the movie plays on the device, various threads control the audio and video in the background.***
- ***Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request.***



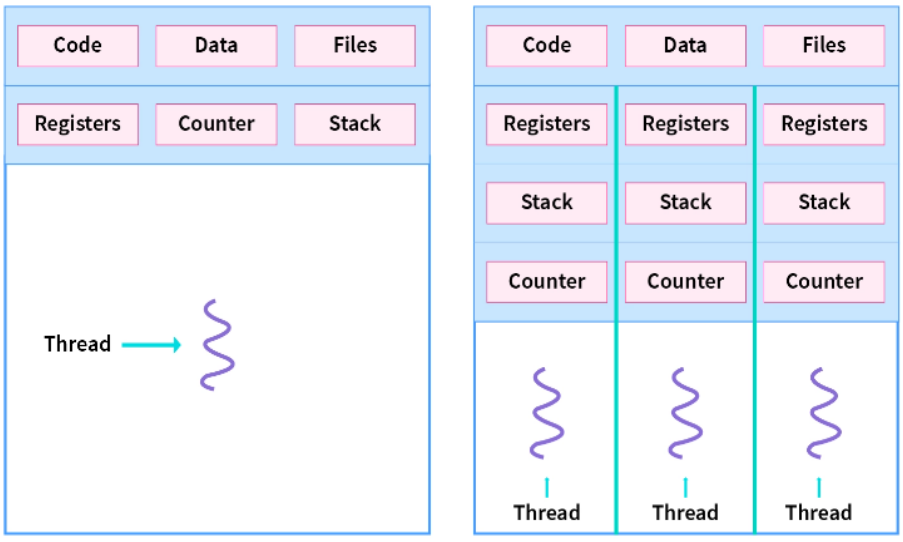
Components of Threads in Operating System

The Threads in Operating System have the following three components.

- Stack Space
- Register Set
- Program Counter

The given below figure shows the working of a single-threaded and a multithreaded process:

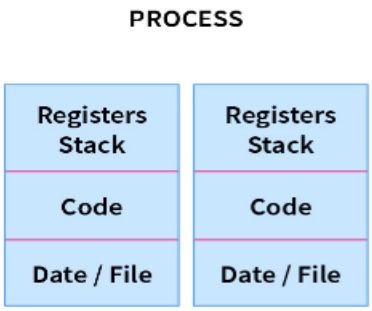
A **single-threaded process** is a process with a single thread. A **multi-threaded process** is a process with multiple threads. As the diagram clearly shows that the multiple threads in it have its own registers, stack, and counter but they share the code and data segment.



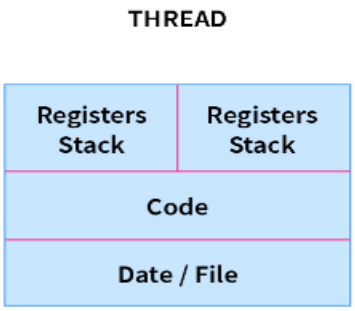
Single-threaded process Multithreaded process

Process simply means any program in execution while the thread is a segment of a process. The main differences between process and thread are mentioned below:

Process	Thread
A Process simply means any program in execution.	Thread simply means a segment of a process.
The process consumes more resources	Thread consumes fewer resources.
The process requires more time for creation.	Thread requires comparatively less time for creation than process.
The process is a heavyweight process	Thread is known as a lightweight process
The process takes more time to terminate	The thread takes less time to terminate.
Processes have independent data and code segments	A thread mainly shares the data segment, code segment, files, etc. with its peer threads.
The process takes more time for context switching.	The thread takes less time for context switching.
Communication between processes needs more time as compared to thread.	Communication between threads needs less time as compared to processes.
For some reason, if a process gets blocked then the remaining processes can continue their execution	In case if a user-level thread gets blocked, all of its peer threads also get blocked.
g: Opening two different browsers.	Eg: Opening two tabs in the same browser.



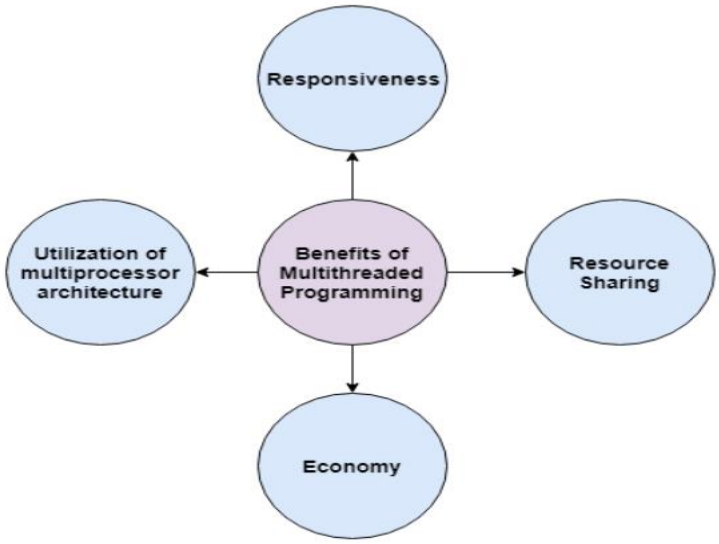
Two Different Process



Two Threads of a single process

Benefits

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



- **Resource Sharing**

Processes may share resources only through techniques such as-Message Passing,Shared Memory

Such techniques must be explicitly organized by programmer. However, threads share the memory and the resources of the process to which they belong by default. A single application can have different threads within the same address space using resource sharing.

- **Responsiveness**

Program responsiveness allows a program to run even if part of it is blocked using multithreading. This can also be done if the process is performing a lengthy operation. For example - A web browser with multithreading can use one thread for user contact and another for image loading at the same time.

- **Utilization of Multiprocessor Architecture**

In a multiprocessor architecture, each thread can run on a different processor in parallel using multithreading. This increases concurrency of the system. This is in direct contrast to a single processor system, where only one process or thread can run on a processor at a time.

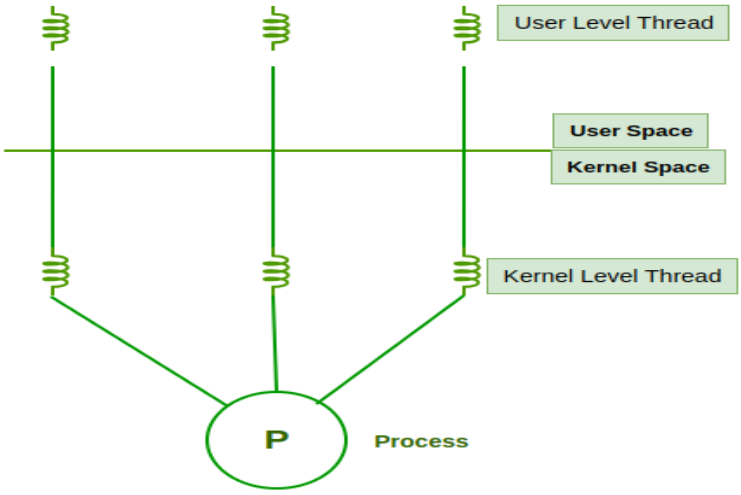
- **Economy**

  It is more economical to use threads as they share the process resources. Comparatively, it is more expensive and time-consuming to create processes as they require more memory and resources. The overhead for process creation and management is much higher than thread creation and management.

Thread Types

Threads are of two types. These are described below.

- User Level Thread
- Kernel Level Thread

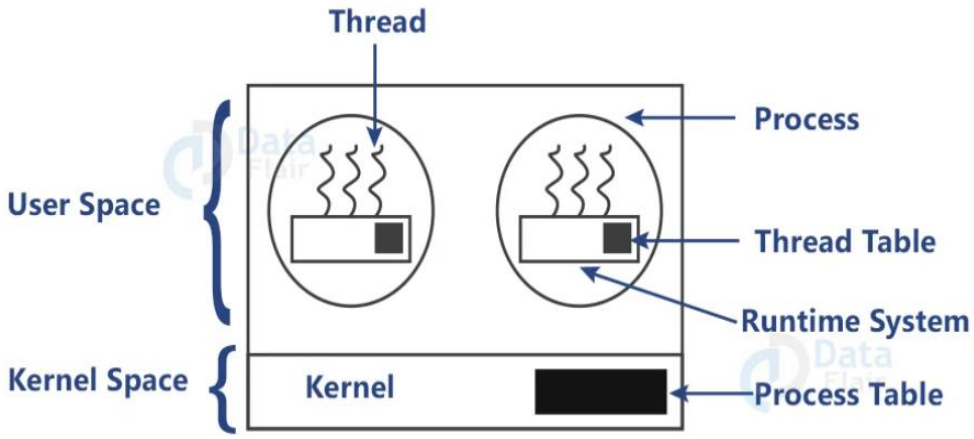


Threads

User Level Threads

User Level Thread is a type of thread that is not created using system calls. The kernel has no work in the management of user-level threads. User-level threads can be easily implemented by the user. In case when user-level threads are single-handed processes, kernel-level thread manages them.

examples: Java thread, POSIX threads, etc.



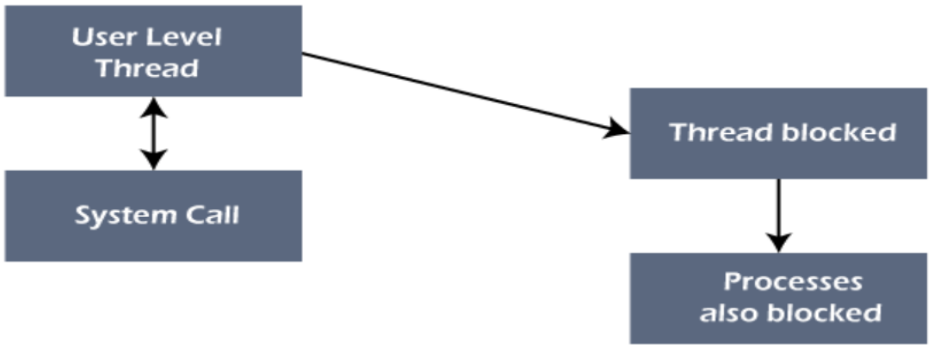
Advantages of User-Level Threads

- Implementation of the User-Level Thread is easier than Kernel Level Thread.

- Context Switch Time is less in User Level Thread.
- User-Level Thread is more efficient than Kernel-Level Thread.
- Because of the presence of only Program Counter, Register Set, and Stack Space, it has a simple representation.

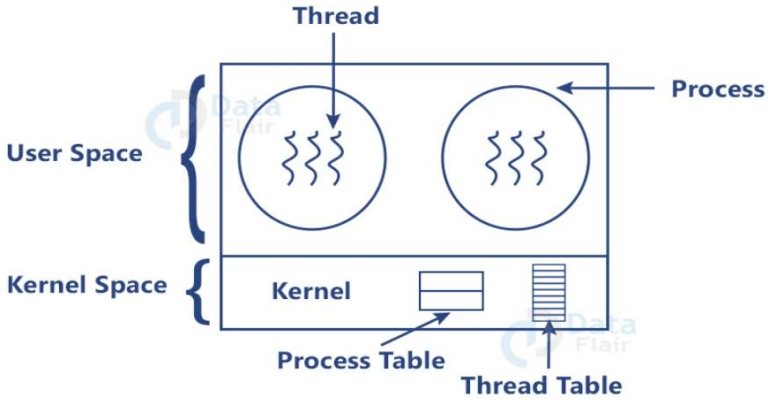
Disadvantages of User-Level Threads

- There is a lack of coordination between Thread and Kernel.
- Inc case of a page fault, the whole process can be blocked.



Kernel Level Threads

A kernel Level Thread is a type of thread that can recognize the Operating system easily. Kernel Level Threads has its own thread table where it keeps track of the system. The operating System Kernel helps in managing threads. Kernel Threads have somehow longer context switching time. Kernel helps in the management of threads. Example: Window Solaris.



Advantages of Kernel-Level Threads

- It has up-to-date information on all threads.
- Applications that block frequently are to be handled by the Kernel-Level Threads.

- Whenever any process requires more time to process, Kernel-Level Thread provides more time to it.

Disadvantages of Kernel-Level threads

- Kernel-Level Thread is slower than User-Level Thread.
- Implementation of this type of thread is a little more complex than a user-level thread.

User Level threads Vs Kernel Level Threads

User level Threads	**Kernel Level Threads**
User thread are implemented by Users	Kernal threads are implemented by OS
OS doesn't recognise user level threads	Kernel threads are recognized by OS
Implementation is easy	Implementation is complicated
Context switch time is less	Context switch time is more
Context switch – no hardware support	hardware support is needed
If one user level thread perform blocking operation then entire process will be blocked	If one kernel level thread perform blocking operation then another thread can continue execution.

There are also hybrid models that combine elements of both user-level and kernel-level threads. For example, some operating systems use a hybrid model called the "two-level model", where each process has one or more user-level threads, which are mapped to kernel-level threads by the operating system.

Advanatges

- Hybrid models combine the advantages of user-level and kernel-level threads, providing greater flexibility and control while also improving performance.
- Hybrid models can scale to larger numbers of threads and processors, which allows for better use of available resources.

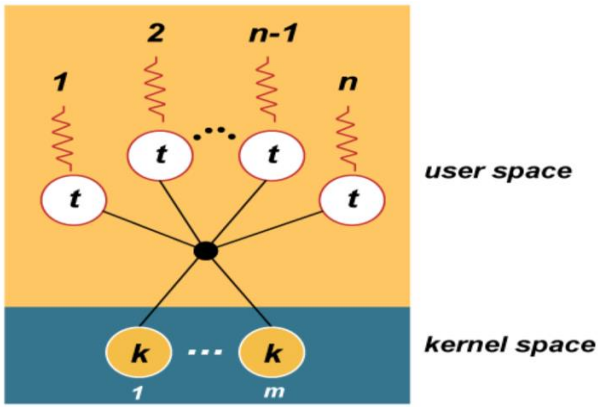
Disadvantages:

- Hybrid models are more complex than either user-level or kernel-level threading, which can make them more difficult to implement and maintain.
- Hybrid models require more resources than either user-level or kernel-level threading, as they require both a thread library and kernel-level support.

User threads are mapped to kernel threads by the threads library. The way this mapping is done is called the* thread model*. Multi threading model are of three types.

Many to Many Model

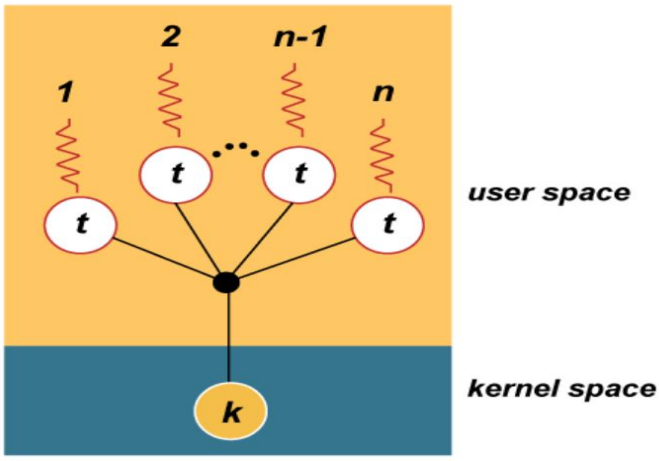
In this model, we have multiple user threads multiplex to same or lesser number of kernel level threads. Number of kernel level threads are specific to the machine, advantage of this model is if a user thread is blocked we can schedule others user thread to other kernel thread. Thus, System doesn't block if a particular thread is blocked. ***It is the best multi threading model.***



Many to One Model

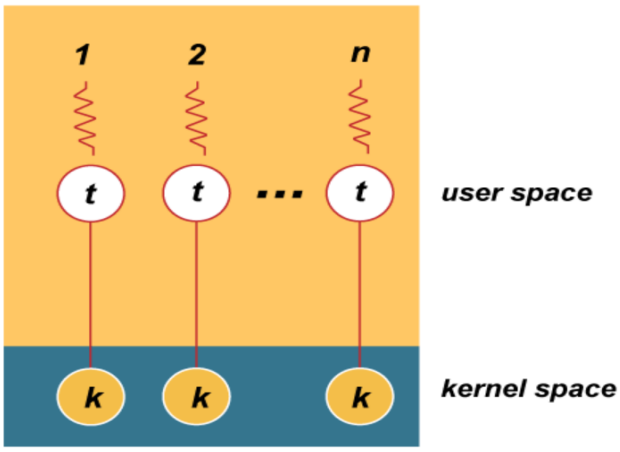
In this model, we have multiple user threads mapped to one kernel thread. In this model when a user thread makes a blocking system call entire process blocks. As we have only one kernel thread and only one user thread can access kernel at a time, so multiple threads are not able access multiprocessor at the same time. The thread management is done on the user level so it is more

efficient.



One to One Model

In this model, one to one relationship between kernel and user thread. In this model multiple thread can run on multiple processor. Problem with this model is that creating a user thread requires the corresponding kernel thread. As each user thread is connected to different kernel , if any user thread makes a blocking system call, the other user threads won't be blocked.



Threading Issues in OS

In a multithreading environment, there are many threading-related problems. Such as

- System Call
- Thread Cancellation
- Signal Handling
- Thread Specific Data
- Thread Pool

- Schedular Activation

fork() and exec() System Calls

- Discussing fork() system call, Let us assume that one of the threads belonging to a multi-threaded program has instigated a fork() call. Therefore, the new process is a duplication of fork(). Here, the question is as follows; will the new duplicate process made by fork() be multi-threaded like all threads of the old process or it will be a unique thread?
- Now, certain UNIX systems have two variants of fork(). fork can either duplicate all threads of the parent process to a child process or just those that were invoked by the parent process. The application will determine which version of fork() to use.
- When the next system call, namely exec() system call is issued, it replaces the whole programming with all its threads by the program specified in the exec() system call's parameters. Ordinarily, the exec() system call goes into queue after the fork() system call.
- However, this implies that the exec() system call should not be queued immediately after the fork() system call because duplicating all the threads of the parent process into the child process will be superfluous. Since the exec() system call will overwrite the whole process with the one given in the arguments passed to exec().This means that in cases like this; a fork() which only replicates one invoking thread will do.

Thread Cancellation

- Thread Cancellation is the task of terminating a thread before it has completed.
- ***For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled.***
- ***Another situation might occur when a user presses a button on a Web browser that stops a Web page from loading any further. Often, a Web page is loaded using several threads-each image is loaded in a separate thread. When a user presses the stop button on the browser, all threads loading the page are cancelled.***
- A thread that is to be cancelled is often referred to as target thread.The Cancellation of a target thread may occur in two different scenarios:

Asynchronous cancellation: One thread immediately terminates the target thread.

Deferred Cancellation : The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

- The difficulty with cancellation occurs in situations where resources have been allocated to a cancelled thread or where a thread is cancelled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a cancelled thread but will not reclaim all resources. Therefore, cancelling a thread asynchronously may not free a necessary system-wide resource.
- With deferred cancellation, in contrast, one thread indicates that a target thread is to be cancelled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be cancelled. The thread can perform this check at a point at which it can be cancelled safely known as cancellation points.

Signal Handling

- Signal Handling is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signalled.
- All signals, whether synchronous or asynchronous, follow the same pattern:

A signal is generated by the occurrence of a particular event.

A generated signal is delivered to a process.

Once delivered, the signal must be handled.

- Examples of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).
- When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as) and having a timer expire. Typically, an asynchronous signal is sent to another process.
- A signal may be handled by one of two possible handlers:

A default signal handler

A user-defined signal handler

- Every signal has a default signal handler that is run by the kernel when handling that signal. This default action can be overridden by a user defined signal handler that is called to handle the signal.
- Signals are handled in different ways. Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.
- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads.
- In general the following options exist:

Deliver the signal to the thread to which the signal applies.

Deliver the signal to every thread in the process.

Deliver the signal to certain threads in the process.

Assign a specific thread to receive all signals for the process.

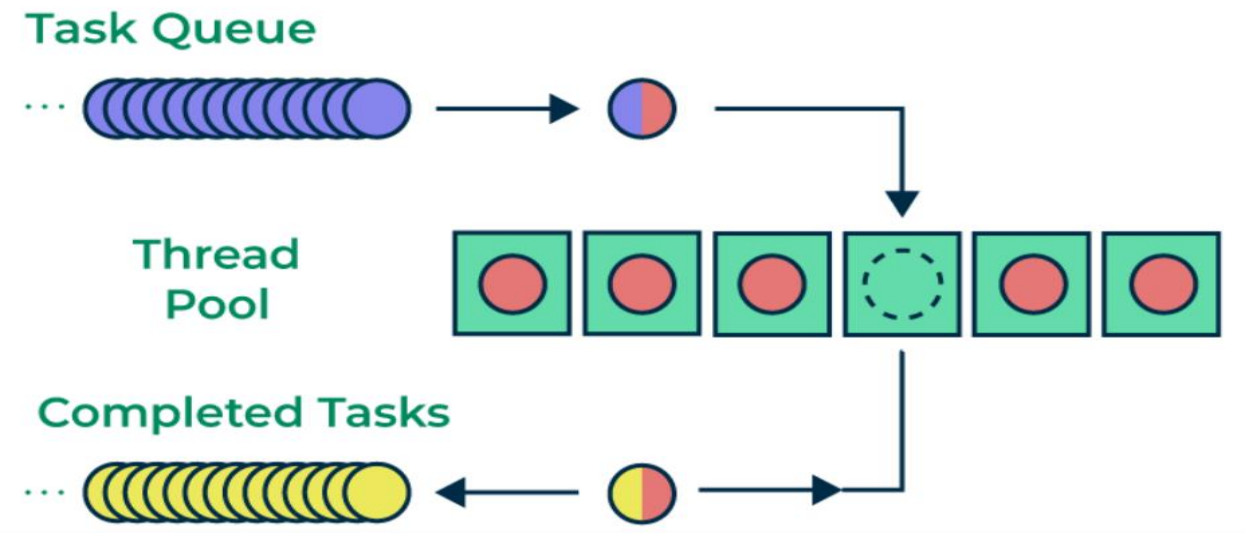
Thread-Specific Data

- Threads belonging to a process share the data of the process. Indeed, this sharing of data provides one of the benefits of multithreaded programming. However, in some circumstances, each thread need its own copy of certain data. We will call such data thread specific data.
- For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specific data.

Thread Pool

- The server develops an independent thread every time an individual attempts to access a page on it. However, the server also has certain challenges. Bear in mind that no limit in the number of active threads in the system will lead to exhaustion of the available system resources because we will create a new thread for each request.
- The establishment of a fresh thread is another thing that worries us. The creation of a new thread should not take more than the amount of time used up by the thread in dealing with the request and quitting after because this will be wasted CPU resources.

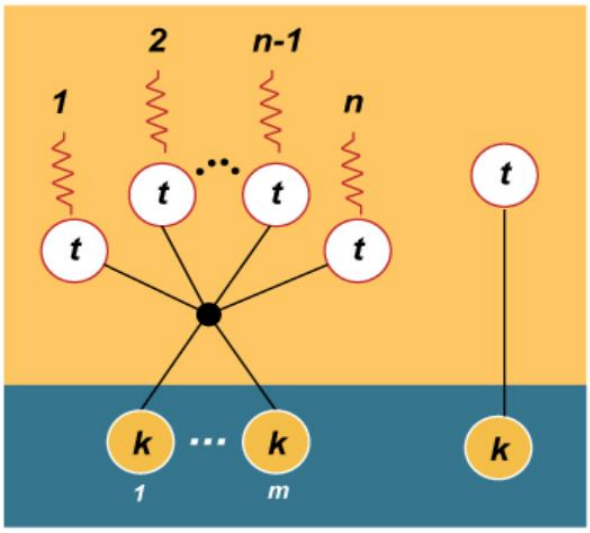
- Hence, thread pool could be the remedy for this challenge. The notion is that as many fewer threads as possible are established during the beginning of the process. A group of threads that forms this collection of threads is referred as a thread pool. There are always threads that stay on the thread pool waiting for an assigned request to service.



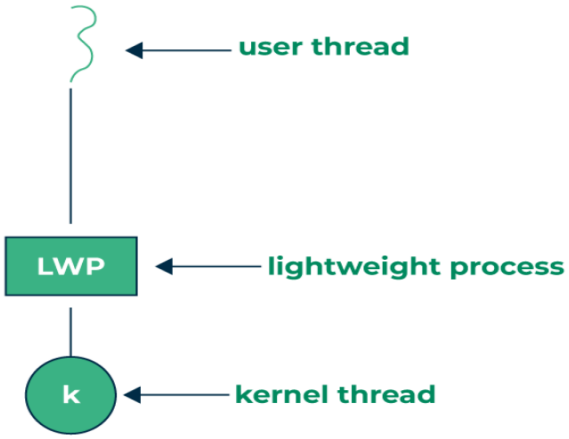
- A new thread is spawned from the pool every time an incoming request reaches the server, which then takes care of the said request. Having performed its duty, it goes back to the pool and awaits its second order.
- Whenever the server receives the request, and fails to identify a specific thread at the ready thread pool, it may only have to wait until some of the threads are available at the ready thread pool. It is better than starting a new thread whenever a request arises because this system works well with machines that cannot support multiple threads at once.

Schedular Activation

- A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required by the many-to-many and two-level models .



Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure-typically known as a lightweight process, or LWP-is shown in Figure.



- To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors.
- If a kernel thread blocks (such as while waiting for an I/0 operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks.
- An application may require any number of LWPs to run efficiently. Consider a CPU-bound application running on a single processor. In this scenario, only one thread can run at once, so one LWP is sufficient. An application that is I/O intensive may require multiple LWPs to execute. Typically, an LWP is required for each concurrent blocking system call. Suppose, for example, that five different file-read requests occur simultaneously. Five LWPs are needed, because all could be waiting for I/0 completion

in the kernel. If a process has only four LWPs, then the fifth request must wait for one of the LWPs to return from the kernel.

- Furthermore, the kernel must inform an application about certain events. This procedure is known as an Upcall. Upcalls are handled by the thread library with an upcall handlers and upcall handlers must run on a virtual processor. One event that triggers an upcall, occurs when an application thread is about to block. In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread.
- The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on the new virtual processor.
- When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. The up call handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor.

Process Scheduling

To increase CPU utilization, multiple processes are loaded into the memory of the CPU and a process is selected from these processes. Loading multiple processes into the main memory is called multiprogramming and the act of determining which process is in the **ready** state, and should be moved to the **running** state is known as **Process Scheduling**.

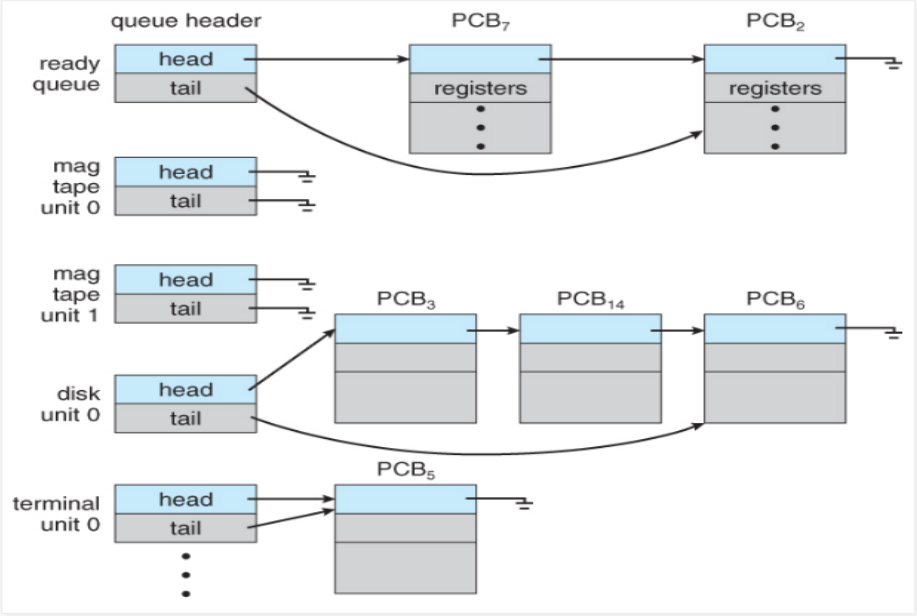
The goal of process scheduling is to achieve efficient utilization of the CPU, fast response time, and fair allocation of resources among all processes.

Process Scheduling Queues

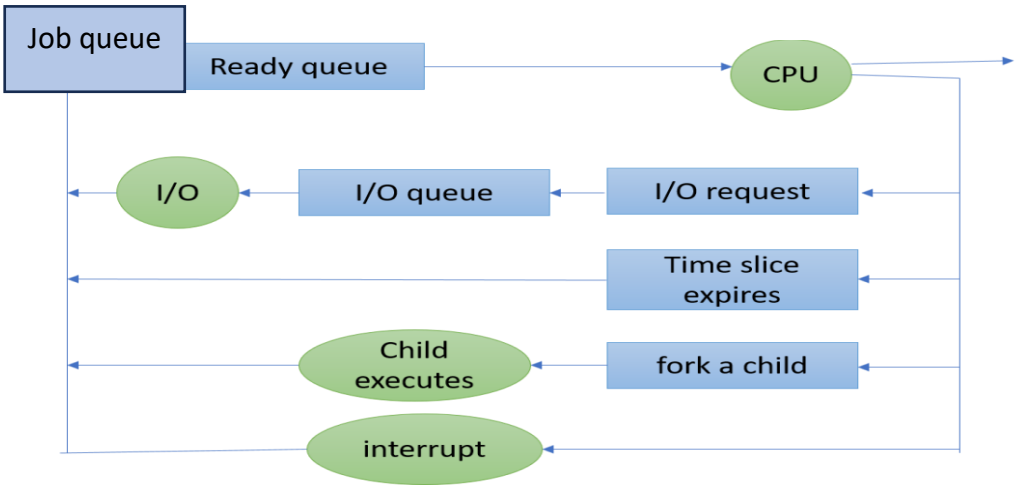
There are multiple states a process has to go through during execution. The OS maintains a separate queue for each state along with the process control blocks (PCB) of all processes. The PCB moves to a new state queue, after being unlinked from its current queue, when the state of a process changes.

These process scheduling queues are:

1. ***Job Queue*** **–** In starting, all the processes get stored in the job queue. It is maintained in the secondary memory.
2. ***Ready Queue*** **–** This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue. Ready queue is maintained in primary memory..
3. ***Device Queue*** **–** When the process needs some IO operation in order to complete its execution, OS changes the state of the process from running to waiting. The context (PCB) associated with the process gets stored on the waiting queue which will be used by the Processor when the process finishes the IO.Each IO Device will have a separate device Queue



A queuing diagram as shown in the figure below represents the scheduling process. There are Two queues – ready queue and device queue. A new process is first admitted into the ready queue. It waits until it is **dispatched** (allocated to the CPU for execution).:



1. The process terminates normally and it exits the system
2. The process issues a I/O request and moves into the device queue. Once it gets the I/O device, it is back into the ready queue
3. The time quantum of the process expires (remember in a time-sharing system, every process gets a fixed amount of CPU time). Then, the process is added back into the ready queue
4. The process creates a child process and waits for the child process to terminate. Hence, it itself is added to the ready queue)

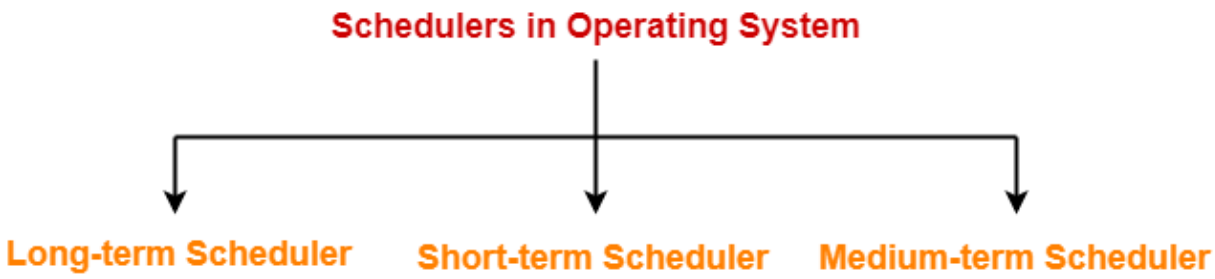
5. A higher priority process interrupts the currently running process. Thus, forcing the current process to change its state to ready state.

Process Schedulers

A scheduler is a special type of system software that handles process scheduling in numerous ways. It mainly selects the jobs that are to be submitted into the system and decides whether the currently running process should keep running or not. If not then which process should be the next one to run. A scheduler makes a decision:

- When the state of the current process changes from running to waiting due to an I/O request .
- If the current process terminates.
- When the scheduler needs to move a process from running to ready state as it has already run for its allotted interval of time.
- When the requested I/O operation is completed, a process moves from the waiting state to the ready state. So, the scheduler can decide to replace the currently-running process with a newly-ready one.
-

There are 3 kinds of schedulers-



1. Long-term Scheduler-

The primary objective of long-term scheduler is to maintain a good degree of multiprogramming

- Long-term scheduler is also known as **Job Scheduler**.
- It selects a balanced mix of I/O bound and CPU bound processes from the secondary memory (new state).

- **CPU Bound Jobs:** CPU-bound jobs are tasks or processes that necessitate a significant amount of CPU processing time and resources (Central Processing Unit). These jobs can put a significant strain on the CPU, affecting system performance and responsiveness.
- **I/O Bound Jobs:** I/O bound jobs are tasks or processes that necessitate a large number of input/output (I/O) operations, such as reading and writing to discs or networks. These jobs are less dependent on the CPU and can put a greater strain on the system's I/O subsystem.
- Then, it loads the selected processes into the main memory (ready state) for execution.

2. Short-term Scheduler-

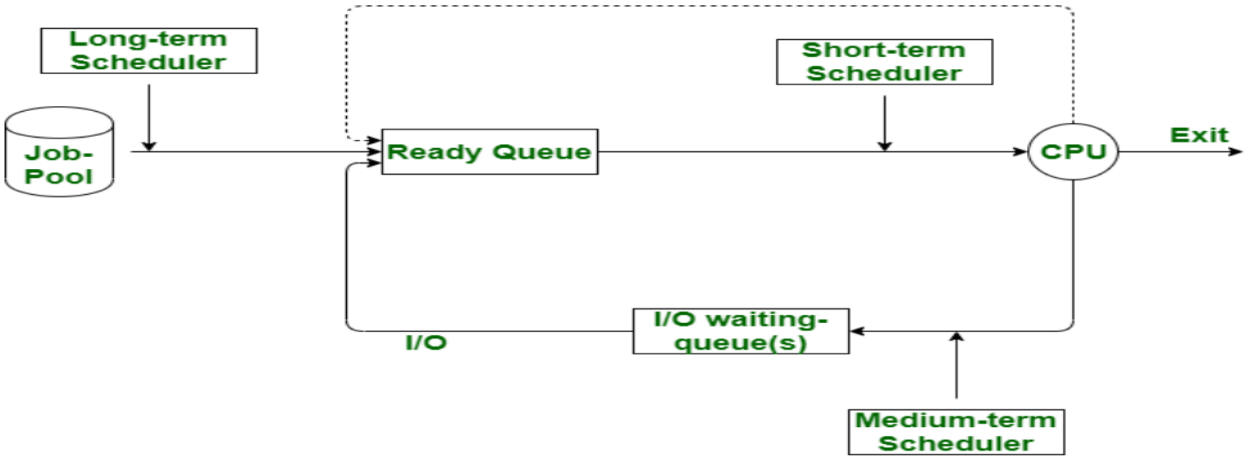
The primary objective of short-term scheduler is to increase the system performance.

- Short-term scheduler is also known as **CPU Scheduler**.
- It decides which process to execute next from the ready queue.
- After short-term scheduler decides the process, **Dispatcher** assigns the decided process to the CPU for execution.
- A dispatcher refers to a module that provides the control of the CPU to that process that gets selected by the short term-scheduler.
- A scheduler is something that helps in selecting a process out of various available processes.

3. Medium-term Scheduler-

The primary objective of medium-term scheduler is to perform swapping.

- Medium-term scheduler swaps-out the processes from main memory to secondary memory to free up the main memory when required.
- Thus, medium-term scheduler reduces the degree of multiprogramming.
- When a running process makes an I/O request it becomes suspended i.e., it cannot be completed. Thus, in order to remove the process from the memory and make space for others, the suspended process is sent to the secondary storage. This is known as swapping, and the process that goes through swapping is said to be swapped out or rolled out.
- After some time when main memory becomes available, medium-term scheduler swaps-in the swapped-out process to the main memory and its execution is resumed from where it left off.



The major differences between long term, medium term and short term scheduler are as follows –

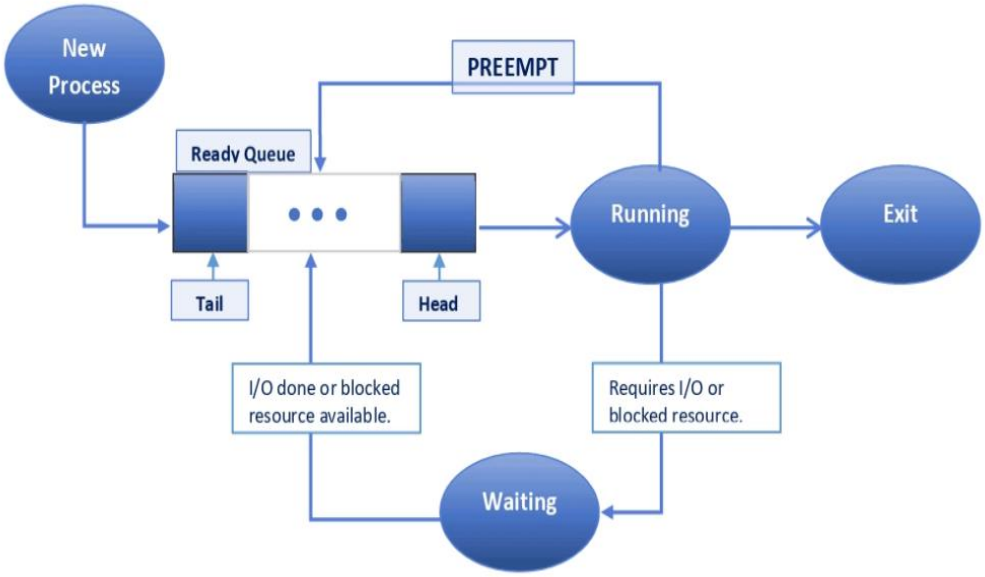
Long term scheduler	**Medium term scheduler**	**Short term scheduler**
Long term scheduler is a job sched	Medium term is a process swapping schedulers.	Short term scheduler is called a scheduler.
The speed of long term is lesser the short term.	The speed of medium term between short and long scheduler.	The speed of short term is fa among the other two.
Long term controls the degre multiprogramming.	Medium term reduces the degr multiprogramming.	The short term provides l control over the degree multiprogramming.
The long term is almost nil or mir in the time sharing system.	The medium term is a part of time sharing system.	Short term is also a minimal sharing system.
The long term selects the proce from the pool and loads them memory for execution.	Medium term can reintroduce process into memory and execu can be continued.	Short term selects those proce that are ready to execute.

CPU Scheduling

CPU Scheduling is a process that allows one process to use the CPU while another process is delayed due to unavailability of any resources such as I / O etc, thus making full use of the CPU. Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

CPU Scheduling can be either Preemptive or Non preemptive

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



- Once the allotted time slice is over or there is a higher-priority process ready for execution (depending on the algorithm), the running process is interrupted, and it switches to the ready state and joins the ready queue.
- Also, whenever a process requires an I/O operation or some blocked resource, it switches to the waiting state. On completion of I/O or receipt of resources, the process switches to the ready state and joins the ready queue.

Preemptive scheduling has a lot of advantages:

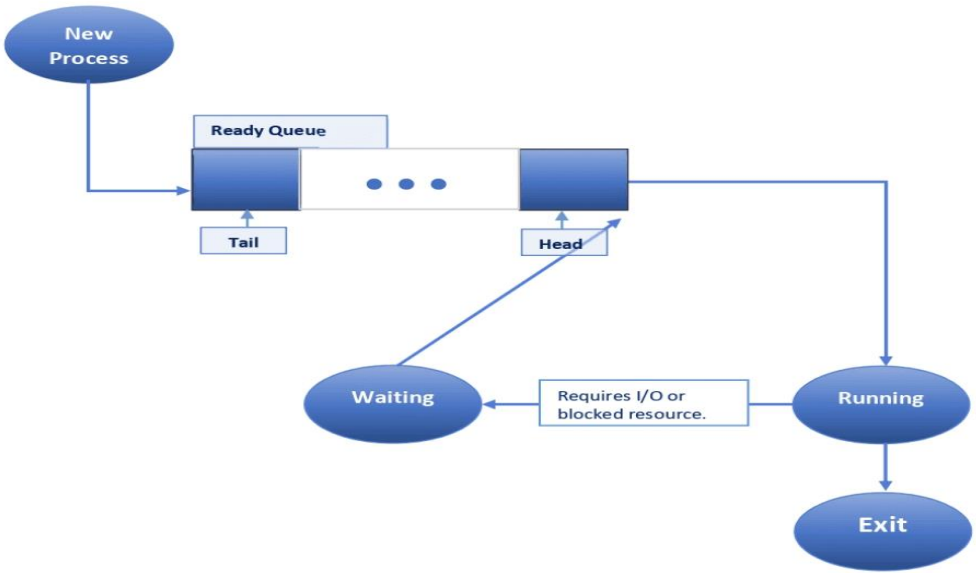
- It ensures no process can monopolize the CPU
- Improves average response time

- Gives room for reconsideration of choice after every interruption, so if priorities change, a higher-priority process can take the CPU
- Avoids deadlocks

It has some disadvantages too:

- Extra time is required for interrupting a running process, switching between contexts, and scheduling newly arriving processes
- **Process starvation** can occur. That's a situation where a low-priority process waits indefinitely or for a long time due to the continuous arrival of higher-priority processes

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



A new process is inserted at the tail of the queue (typically in FCFS). Based on the specific algorithm, a process is selected for execution. The selected process continues until completion without any interruption.

However, if the process requires any I/O operation or some blocked resource while running, it will switch to the waiting state. On completion of the I/O operation or receipt of the needed resource, the process immediately returns to the top of the queue.

Advantage : Non-preemptive scheduling has minimal computational overhead and is very simple to understand and implement.

Significant disadvantage associated with this type of scheduling is the difficulty in handling priority scheduling. This is so because a process cannot be preempted. Let's suppose a less important process

with a long burst time (time needed to complete execution) is running. Then, a critical process (very high priority) arrives in the queue. This becomes a difficult scenario since a running process cannot be interrupted .This may lead to Deadlock.

Scheduling Criteria in an Operating System

There are different *CPU* scheduling algorithms with different properties. The choice of algorithm is dependent on various different factors. There are many criteria suggested for comparing *CPU* schedule algorithms, some of which are:

- *CPU* utilization
- Throughput
- Turnaround time
- Waiting time
- Response time

CPU utilization: The main objective of any CPU scheduling algorithm is to keep the CPU as busy as possible. Theoretically, CPU utilization can range from 0 to 100 but in a real-time system, it varies from 40 to 90 percent depending on the load upon the system.

Throughput:A measure of the work done by the CPU is the number of processes being executed and completed per unit of time. This is called throughput. The throughput may vary depending on the length or duration of the processes.

Arrival time:The arrival time of a process is when a process is ready to be executed, which means it is finally in a ready state and is waiting in a queue for its turn to be executed.

Burst Time: The **burst time** of a process is the number of time units it requires to be executed by the CPU

Completion Time: The completion time is the time when the process stops executing, which means that the process has completed its burst time and is completely executed.

Turnaround time: For a particular process, an important criterion is how long it takes to execute that process. The time elapsed from the time of submission of a process to the time of completion is known as the turnaround time. Turn-around time is the sum of times spent waiting to get into memory, waiting in the ready queue, executing in CPU, and waiting for I/O.

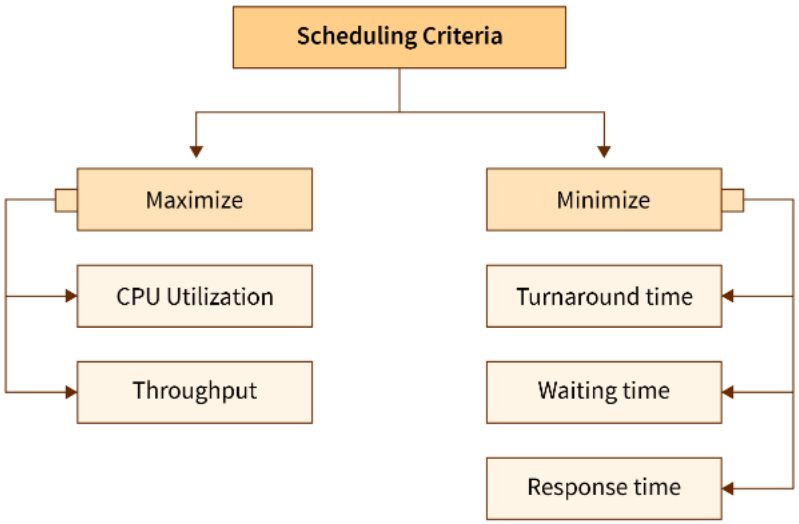
Turn Around Time = Completion Time – Arrival Time.

Waiting time-:A scheduling algorithm does not affect the time required to complete the process once it starts execution. It only affects the waiting time of a process i.e. time spent by a process waiting in the ready queue.

Waiting Time = Turnaround Time – Burst Time.

Response time :In an interactive system, turn-around time is not the best criterion. A process may produce some output fairly early and continue computing new results while previous results are being output to the user. Thus another criterion is the time taken from submission of the process of the request until the first response is produced. This measure is called response time.

Response Time = CPU Allocation Time(when the CPU was allocated for the first) – Arrival Time



The aim of the scheduling algorithm is to maximize and minimize the following:

Maximize:

- **CPU utilization** - It makes sure that the *CPU* is operating at its peak and is busy.
- **Throughoutput** - It is the number of processes that complete their execution per unit of time.

Minimize:

- **Waiting time**- It is the amount of waiting time in the queue.
- **Response time**- Time retired for generating the first request after submission.
- **Turnaround time**- It is the amount of time required to execute a specific process.

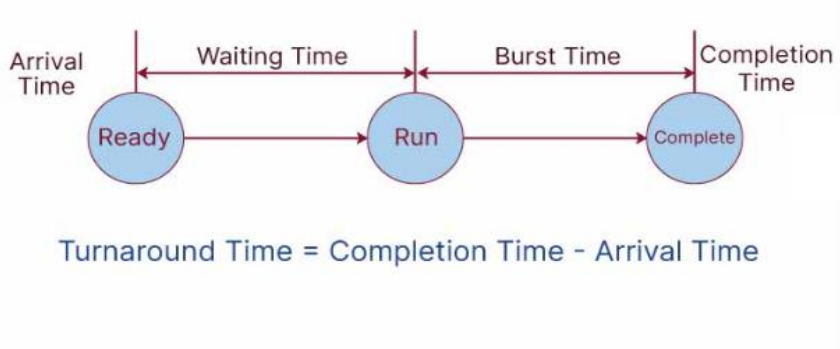
CPU SCHEDULING

Need for CPU Scheduling

In a single-processor system, only one job can be processed at a time; the rest must wait until the CPU gets free and can be rescheduled. Multiprogramming aims to have a process running at all times to maximize CPU utilization. The concept is straightforward: a process runs until it needs to wait, typically for an I/O request to complete. In a simple operating system, the CPU then stands idle. All this waiting time is wasted; no fruitful work can be performed. With multiprogramming, you can use this time to process other jobs productively with the help of Scheduling.

What is CPU Scheduling

CPU Scheduling is the process which determines the process which will own the CPU for execution while other processes are in the queue. Whenever the CPU is idle, the Operating System selects one of the process which is ready in the queue. This task is carried out by CPU scheduler which selects one of the processes in memory that is ready for execution. Scheduling is used to increase the efficiency of CPU. CPU Scheduling is implemented to minimize Waiting Time, Response Time, and turnaround time and maximize CPU utilization and Throughput.

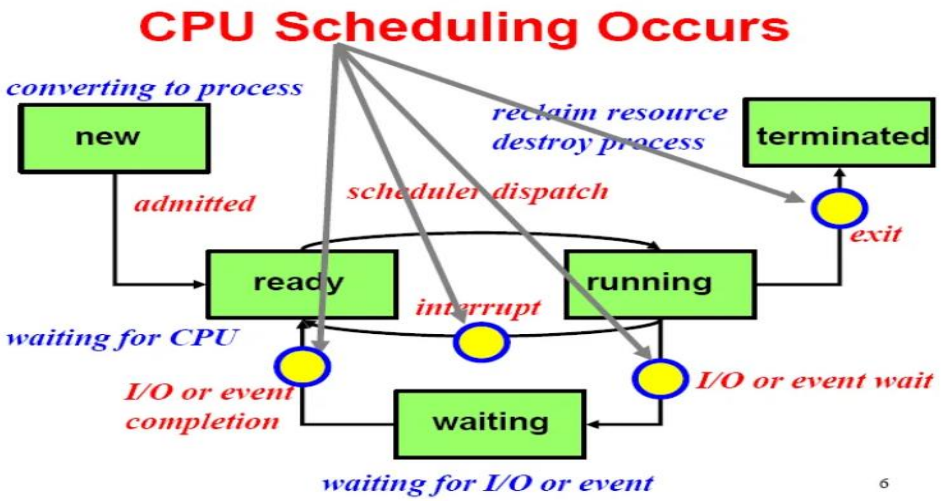


TAT=Completion Time-Arrival Time

Waiting Time=Turn Around Time-Burst Time

Response Time =First Response - Arrival Time

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

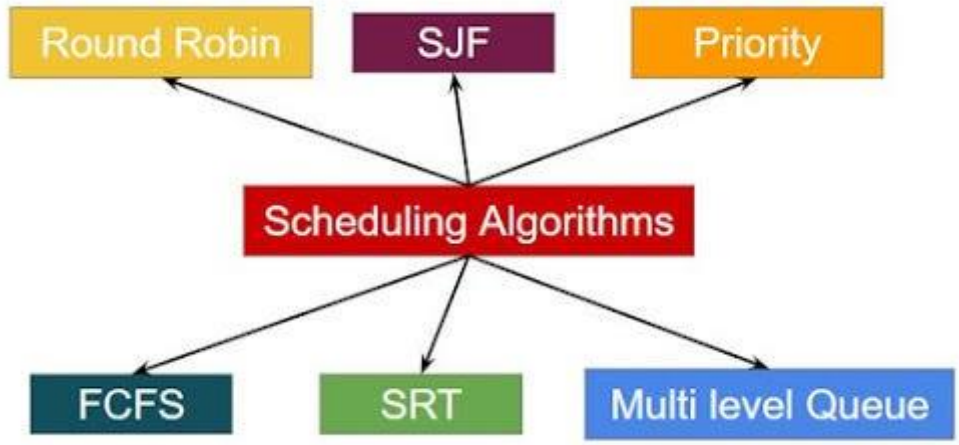


- When a process switches from the running state to the waiting state(for I/O request or invocation of wait for the termination of one of the child processes).
- When a process switches from the running state to the ready state (for example, when an interrupt occurs).
- When a process switches from the waiting state to the ready state(for example, completion of I/O).
- When a process terminates.

Types of CPU Scheduling Algorithm:

There are 6 types of Scheduling Algorithms:

1. FCFS (**F**irst **C**ome **F**irst **S**erve)
2. SJF (**S**hortest **J**ob **F**irst)
3. SRT (**S**hortest **R**emaining **T**ime)
4. RR (**R**ound **R**obin Scheduling)
5. Priority Scheduling
6. Multiple level feedback Scheduling



First Come First Serve Scheduling

It is the easy and simple CPU Scheduling Algorithm. In this algorithm, the process which requests the CPU first, gets the CPU first for execution. It can be implemented using FIFO (First-In, First-Out) Queue method.

A simple real-life example of this algorithm is the cash counter. There is a queue on the counter. The person who arrives first at the counter receives the services first, followed by the second person, then third, and so on. The CPU process also works like this.

Advantages of FCFS:

The following are some benefits of using the FCFS scheduling algorithm:

1. The job that comes first is served first.
2. It is the CPU scheduling algorithm's simplest form.
3. It is quite easy to program.

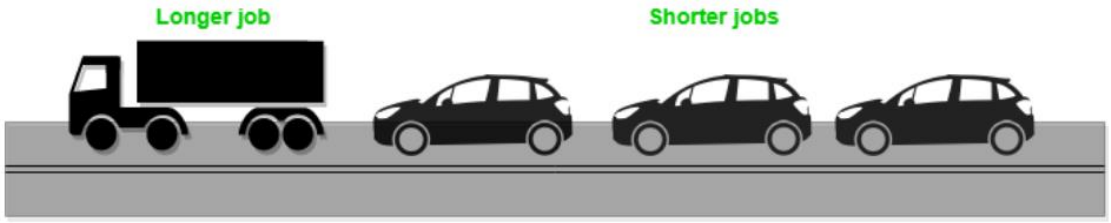
Disadvantages

1. It is **non-pre-emptive** algorithm, which means the **process priority** doesn't matter.

If a process with very least priority is being executed, more like **daily routine backup** process, which takes more time, and all of a sudden, some other high priority process arrives, like **interrupt to avoid system crash**, the high priority process will have to wait, and hence in this case, the system will crash, just because of improper process scheduling.

2. In FCFS, the Average Waiting Time is comparatively high.
3. Resources utilization in parallel is not possible, which leads to **Convoy Effect**, (Convoy Effect is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time and hence poor resource(CPU, I/O etc) utilization.

Convoy Effect: ***Convoy effect is phenomenon associated with the First Come First Serve (FCFS) algorithm, in which the whole Operating System slows down due to few slow processes.***



Example 1 -FCFS Scheduling:(Without Arrival Times)

TAT=Completion Time-Arrival Time

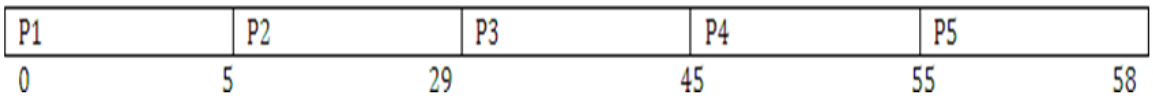
Waiting Time=Turn Around Time-Burst Time

Response Time =First Response - Arrival Time

Process	Burst time(milliseconds)
P1	5
P2	24
P3	16
P4	10
P5	3

Gantt Chart for FCFS: **(Generalized Activity Normalization Time Table (GANTT))**

A Gantt chart is a horizontal bar chart used to represent operating systems CPU scheduling in graphical view that help to plan, coordinate and track specific CPU utilization factor like throughput, waiting time, turnaround time etc.



Average turn around time:

TAT=Completion Time-Arrival Time

Turn around time for p1= 5=0=5.

Turn around time for p2=29-0=29

Turn around time for p3=45-0=45

Turn around time for p4=55-0=55

Turn around time for p5= 55-0=58

Average turn around time= (5+29+45+55+58/5) = 187/5 =37.5 millisecounds

Average waiting time:

Waiting Time=Turn Around Time-Burst Time

Waiting time for p1=5-5=0

Waiting time for p2=29-24=5

Waiting time for p3=45-16=29

Waiting time for p4=55-10=45

Waiting time for p5=58-3=55

Average waiting time= (0+5+29+45+55)/5 = 125/5 = 25 ms.

Average Response Time :

First Response - Arrival Time

Response Time for P1 =0-0=0

Response Time for P2 => 5-0 = 5

Response Time for P3 => 29-0 = 29

Response Time for P4 => 45-0 = 45

Response Time for P5 => 55-0 = 55

Average Response Time => (0+5+29+45+55)/5 =>25ms

Process	Burst time(milliseconds)	Completion Time	Turnaround Time	Waiting Time	Response time
P1	5	5	5	0	0
P2	24	29	29	5	5
P3	16	45	45	29	29
P4	10	55	55	45	45
P5	3	58	58	55	55

First Come FirstServe:(With Arrival Times)

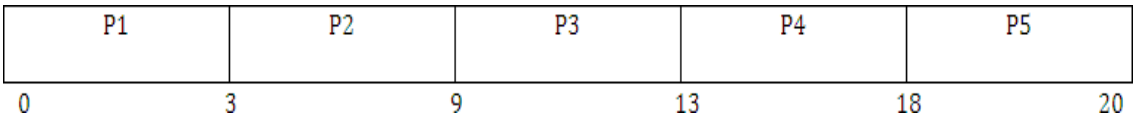
TAT=Completion Time-Arrival Time

Waiting Time=Turn Around Time-Burst Time

First Response - Arrival Time

PROCESS	BURST TIME	ARRIVAL TIME
P1	3	0
P2	6	2
P3	4	4
P4	5	6
P5	2	8

Gantt Chart



Average Turn Around Time

TAT=Completion Time-Arrival Time

Turn Around Time for P1 => 3-0= 3

Turn Around Time for P2 => 9-2 = 7

Turn Around Time for P3 => 13-4=9

Turn Around Time for P4 => 18-6= 12

Turn Around Time for P5 => 20-8=12

Average Turn Around Time => (3+7+9+12+12)/5 =>43/5 = 8.50 ms.

Average Response Time :

Response Time = First Response - Arrival Time

Response Time of P1 = 0

Response Time of P2 => 3-2 = 1

Response Time of P3 => 9-4 = 5

Response Time of P4 => 13-6 = 7

Response Time of P5 => 18-8 =10

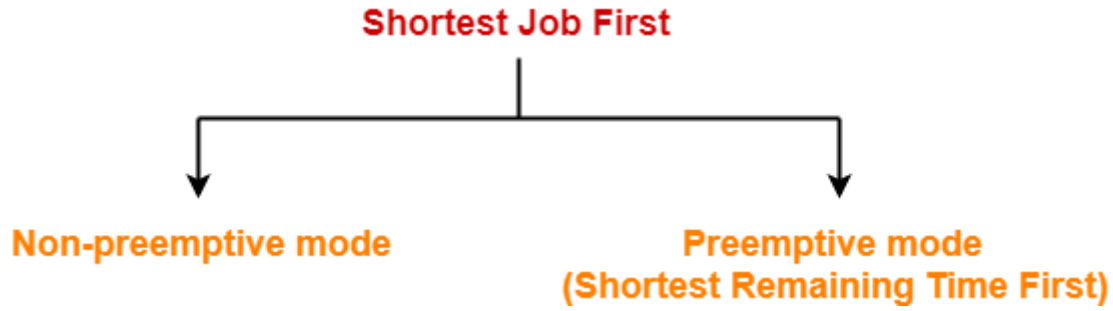
Average Response Time => (0+1+5+7+10)/5 => 23/5 = 4.6 ms

Process	Burst Time	Arrival Time	Completion Time	Turnaround Time	Waiting Time	Response Time
P1	3	0	3	3	0	0
P2	6	2	9	7	1	1
P3	4	4	13	9	5	5
P4	5	6	18	12	7	7
P5	2	8	20	12	10	10

Shortest Job First

Till now, we were scheduling the processes according to their arrival time (in FCFS scheduling). However, SJF scheduling algorithm, schedules the processes according to their burst time. In SJF scheduling, the process with the lowest burst time, among the list of available processes in the ready queue, is going to be scheduled next.

In case of a tie, it is broken by **FCFS Scheduling**.



Non-Premptive Mode(Example)

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4

P3	4	2
P4	0	6
P5	2	3

Gantt Chart

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time/Finishing/Completion Time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 1 = 3
P2	16	16 – 1 = 15	15 – 4 = 11
P3	9	9 – 4 = 5	5 – 2 = 3
P4	6	6 – 0 = 6	6 – 6 = 0
P5	12	12 – 2 = 10	10 – 3 = 7

Now,

- ***Average Turn Around time = (4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 unit***
- ***Average waiting time = (3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 unit***

Advantages

- According to the definition, short processes are executed first and then followed by longer processes.
- The throughput is increased because more processes can be executed in less amount of time.

Disadvantages:

- The time taken by a process must be known by the CPU beforehand, which is not possible.
- Longer processes will have more waiting time, eventually they'll suffer starvation.

Shortest Remaining Time First(SRTF)

The Preemptive version of Shortest Job First(SJF) scheduling is known as Shortest Remaining Time First (SRTF). With the help of the SRTF algorithm, the process having the smallest amount of time remaining until completion is selected first to execute.

Consider the set of 6 processes whose arrival time and burst time are given below

Process Id	Arrival time	Burst time
P1	0	7
P2	1	5
P3	2	3
P4	3	1
P5	4	2
P6	5	1

Gantt Chart-

Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	19	19 – 0 = 19	19 – 7 = 12
P2	13	13 – 1 = 12	12 – 5 = 7
P3	6	6 – 2 = 4	4 – 3 = 1
P4	4	4 – 3 = 1	1 – 1 = 0
P5	9	9 – 4 = 5	5 – 2 = 3
P6	7	7 – 5 = 2	2 – 1 = 1

Advantages

- Processes are executed faster than SJF, being the preemptive version of it.

Disadvantages

- Context switching is done a lot more times and adds to the overhead time.
- Like SJF, it may still lead to starvation and requires the knowledge of process time beforehand.
- Impossible to implement in interactive systems where the required CPU time is unknown.

Priority Scheduling Algorithm

Priority scheduling in OS is the scheduling algorithm that schedules processes according to the priority assigned to each of the processes. Higher priority processes are executed before lower priority process

Priority of processes depends on some factors such as:

- Time limit
- Memory requirements of the process
- Ratio of average I/O to average CPU burst time

There can be more factors on the basis of which the priority of a process/job is determined. This priority is assigned to the processes by the scheduler.These priorities of processes are represented as simple integers in a fixed range such as 0 to 7, or maybe 0 to 4095. These numbers depend on the type of system.

Note : Generally, the process with the Larger integer number will have low priority, and the process with the smaller integer value will have high priority.

Priorities can be static or dynamic, depending on the situation.

Static priority: It doesn't change priority throughout the execution of the process

Dynamic priority: In this dynamic priority, priority can be changed by the scheduler at a regular interval of time.

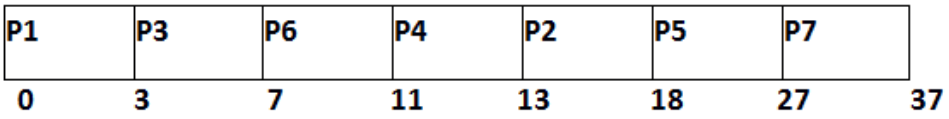
Priority scheduling can be of two types:

1. **Preemptive Priority Scheduling**: If the new process arrived at the ready queue has a higher priority than the currently running process, the CPU is preempted, which means the processing of the current process is stoped and the incoming new process with higher priority gets the CPU for its execution.
2. **Non-Preemptive Priority Scheduling**: In case of non-preemptive priority scheduling algorithm if a new process arrives with a higher priority than the current running process, the incoming process is put at the head of the ready queue, which means after the execution of the current process it will be processed.

Non-Preemptive Priority Scheduling

Process ID	Priority	Arrival Time	Burst Time
1	2	0	3
2	6	2	5
3	3	1	4
4	5	4	2
5	7	6	9
6	4	5	4

7	10	7	10



process Id	Priority	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time	Response Time
1	2	0	3	3	3	0	0
2	6	2	5	18	16	11	13
3	3	1	4	7	6	2	3
4	5	4	2	13	9	7	11
5	7	6	9	27	21	12	18
6	4	5	4	11	6	2	7
7	10	7	10	37	30	18	27

Avg Waiting Time = (0+11+2+7+12+2+18)/7 = 52/7 units

Preemptive Priority CPU Scheduling

Step-1: Select the first process whose arrival time will be 0, we need to select that process because that process is only executing at time t=0.

Step-2: Check the priority of the next available process. Here we need to check for 3 conditions.

if **priority(current_process) > priority(prior_process)** :- then execute the current process.

if **priority(current_process) < priority(prior_process)** :- then execute the prior process.

if **priority(current_process) = priority(prior_process)** :- then execute the process which arrives first i.e., arrival time should be first.

Step-3: Repeat **Step-2** until it reaches the final process.

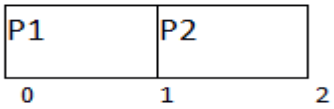
Step-4: When it reaches the final process, choose the process which is having the highest priority & execute it. Repeat the same step until all processes complete their execution.

Process Id	Priority	Arrival Time	Burst Time
1	2(L)	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10(H)	5	15
7	9	15	8

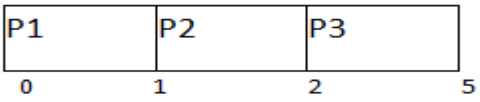
GANTT chart Preparation

At time 0, P1 arrives with the burst time of 1 units and priority 2. Since no other process is available hence this will be scheduled till next job arrives or its completion (whichever is lesser).

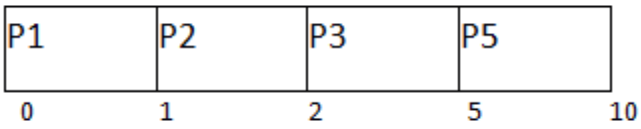
At time 1, P2 arrives. P1 has completed its execution and no other process is available at this time hence the Operating system has to schedule it regardless of the priority assigned to it.



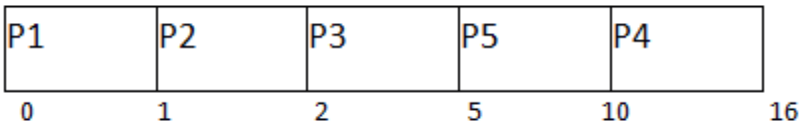
The Next process P3 arrives at time unit 2, the priority of P3 is higher to P2. Hence the execution of P2 will be stopped and P3 will be scheduled on the CPU.



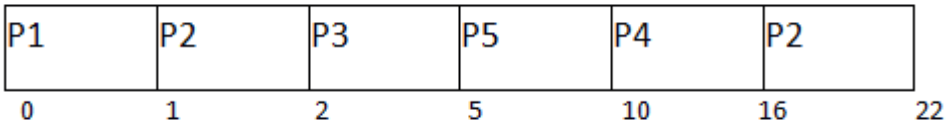
During the execution of P3, three more processes P4, P5 and P6 becomes available. Since, all these three have the priority lower to the process in execution so P3 can't preempt the process. P3 will complete its execution and then P5 will be scheduled with the priority highest among the available processes.



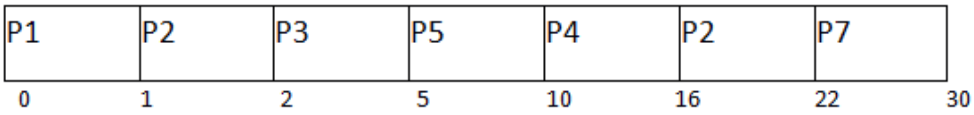
Meanwhile the execution of P5, all the processes got available in the ready queue. At this point, the algorithm will start behaving as Non Preemptive Priority Scheduling. Hence now, once all the processes get available in the ready queue, the OS just took the process with the highest priority and execute that process till completion. In this case, P4 will be scheduled and will be executed till the completion.



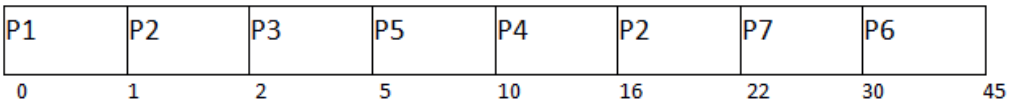
Since P4 is completed, the other process with the highest priority available in the ready queue is P2. Hence P2 will be scheduled next.



P2 is given the CPU till the completion. Since its remaining burst time is 6 units hence P7 will be scheduled after this.



The only remaining process is P6 with the least priority, the Operating System has no choice unless of executing it. This will be executed at the last.



The Completion Time of each process is determined with the help of GANTT chart. The turnaround time and the waiting time can be calculated by the following formula.

1. Turnaround Time = Completion Time - Arrival Time
2. Waiting Time = Turn Around Time - Burst Time

Process Id	Priority	Arrival Time	Burst Time	Completion Time	Turn around Time	Waiting Time
1	2	0	1	1	1	0
2	6	1	7	22	21	14
3	3	2	3	5	3	0
4	5	3	6	16	13	7
5	4	4	5	10	6	1
6	10	5	15	45	40	25
7	9	6	8	30	24	16

Avg Waiting Time = (0+14+0+7+1+25+16)/7 = 63/7 = 9 units

Advantages of priority scheduling in OS

- Easy to use.
- Processes with higher priority execute first which saves time.
- The importance of each process is precisely defined.
- A good algorithm for applications with fluctuating time and resource requirements.

Disadvantages of priority scheduling in OS

- We can lose all the low-priority processes if the system crashes.
- This process can cause starvation if high-priority processes take too much CPU time. The lower priority process can also be postponed for an indefinite time.

Ageing Technique

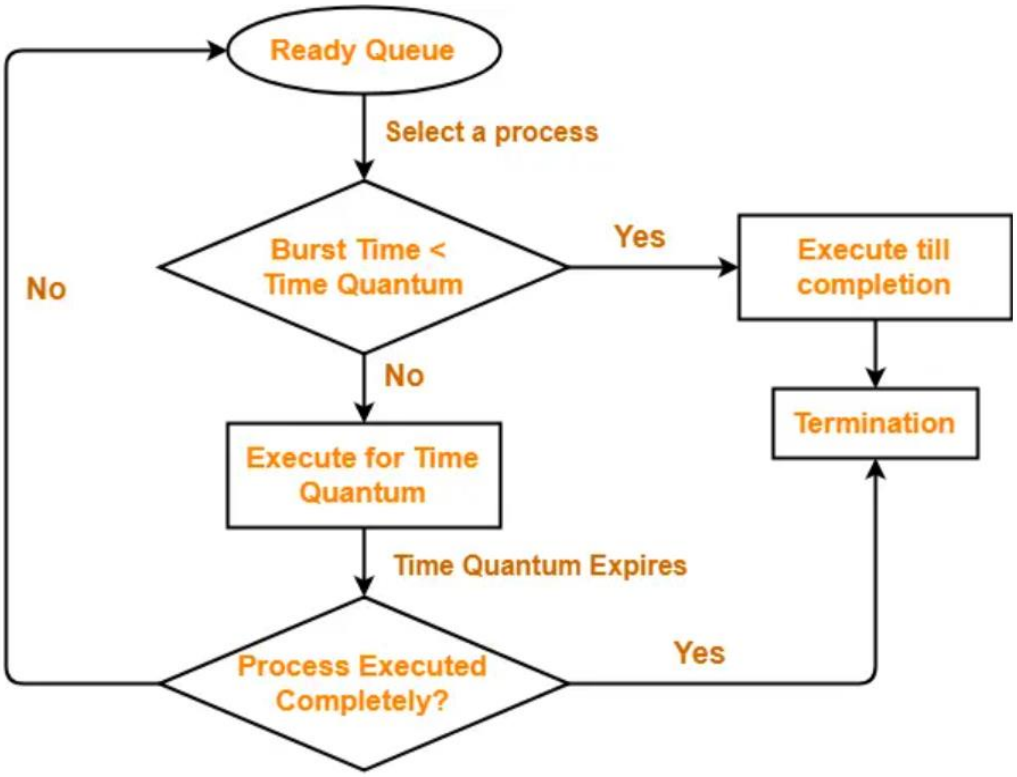
Ageing technique can help prevent starvation of a process. In this, we can increase the priority of the low-priority processes based on their waiting time. This ensures that no process waits for an indefinite time to get CPU time.

Round Robin Scheduling,

In Round Robin Scheduling,

- CPU is assigned to the process on the basis of FCFS for a fixed amount of time.
- This fixed amount of time is called as **time quantum** or **time slice**.
- After the time quantum expires, the running process is preempted and sent to the ready queue.
- Then, the processor is assigned to the next arrived process.
- It is always preemptive in nature.

> Round Robin Scheduling is FCFS Scheduling with preemptive mode.



Round-robin scheduling algorithm with an example.

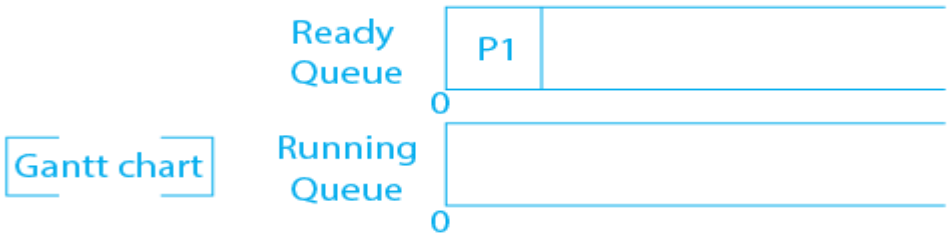
Let's see how the round-robin scheduling algorithm works with an example. Here, we have taken an example to understand the working of the algorithm. We will also do a dry run to understand it better.

Process No	Arrived Time	Brust Time
P1	0	5
P2	1	4
P3	2	2
P4	4	1

In the above example, we have taken 4 processes P1, P2, P3, and P4 with an arrival time of 0,1,2, and 4 respectively. They also have burst times 5, 4, 2, and 1 respectively. Now, we need to create two queues the ready queue and the running queue which is also known as the **Gantt chart.**

Step 1: first, we will push all the processes in the ready queue with an arrival time of 0. In this example, we have only P1 with an arrival time of 0.

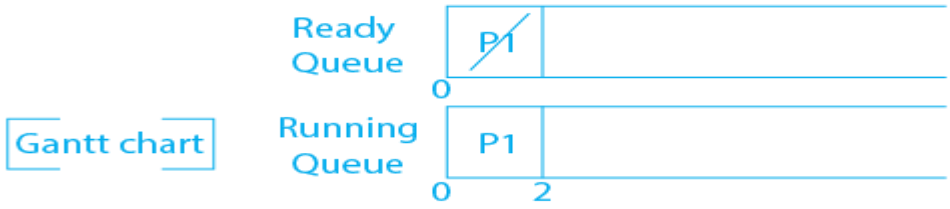
Process No	Arrived Time	Brust Time
P1	0	5
P2	1	4
P3	2	2
P4	4	1



This is how queues will look after the completion of the first step.

Step 2: Now, we will check in the ready queue and if any process is available in the queue then we will remove the first process from the queue and push it into the running queue. Let's see how the queue will be after this step.

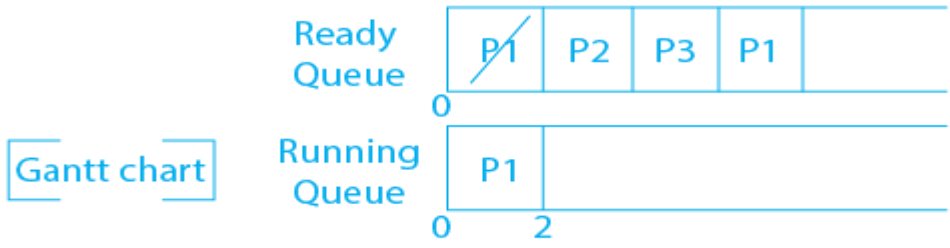
Process No	Arrived Time	Brust Time
P1	0	3
P2	1	4
P3	2	2
P4	4	1



In the above image, we can see that we have pushed process P1 from the ready queue to the running queue. We have also decreased the burst time of process P1 by 2 units as we already executed 2 units of P1.

Step 3: Now we will push all the processes arrival time within 2 whose burst time is not 0.

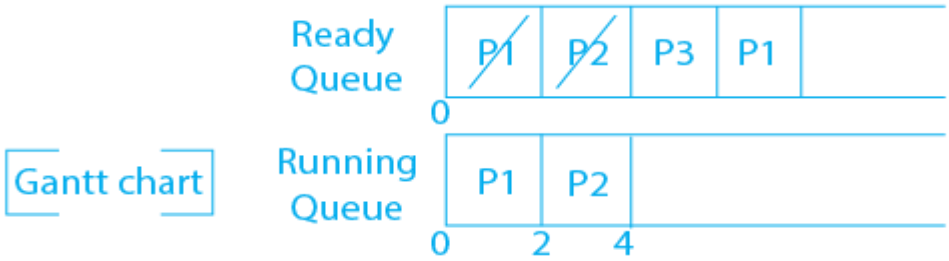
Process No	Arrived Time	Brust Time
P1	0	3
P2	1	4
P3	2	2
P4	4	1



In the above image, we can see that we have two processes with an arrival time within 2 P2 and P3 so, we will push both processes into the ready queue. Now, we can see that process P1 also has remaining burst time so we will also push process P1 into the ready queue again.

Step 4: Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.

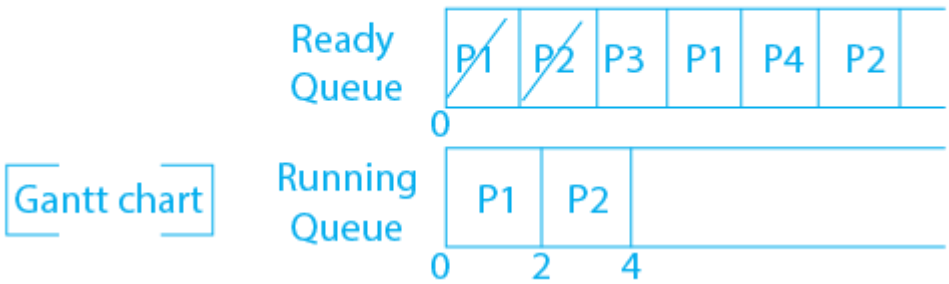
Process No	Arrived Time	Brust Time
P1	0	3
P2	1	4
P3	2	2
P4	4	1



In the above image, we can see that we have pushed process P2 from the ready queue to the running queue. We also decreased the burst time of the process P2 as it already executed 2 units.

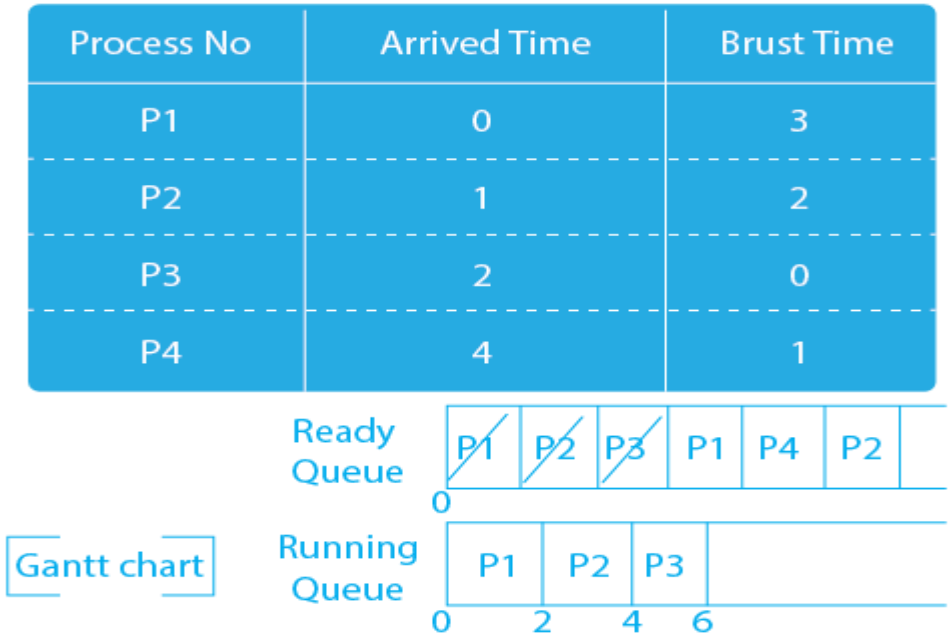
Step 5: Now we will push all the processes arrival time within 4 whose burst time is not 0.

Process No	Arrived Time	Brust Time
P1	0	3
P2	1	2
P3	2	2
P4	4	1



In the above image, we can see that we have one process with an arrival time within 4 P4 so, we will push that process into the ready queue. Now, we can see that process P2 also has remaining burst time so we will also push process P2 into the ready queue again.

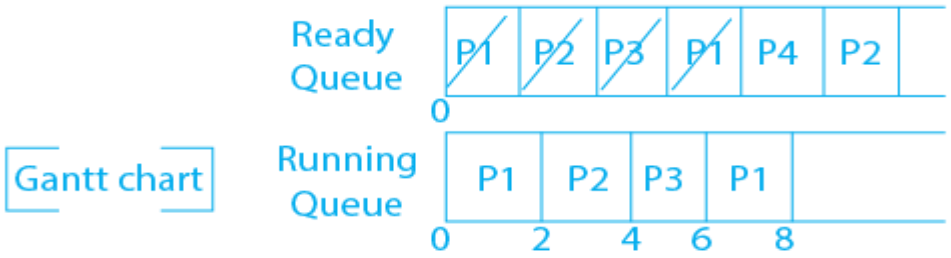
Step 6: Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.



In the above image, we can see that we have pushed process P3 from the ready queue to the running queue. We also decreased the burst time of the process P3 as it already executed 2 units. Now, process P3's burst time becomes 0 so we will not consider it further.

Step 7: Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.

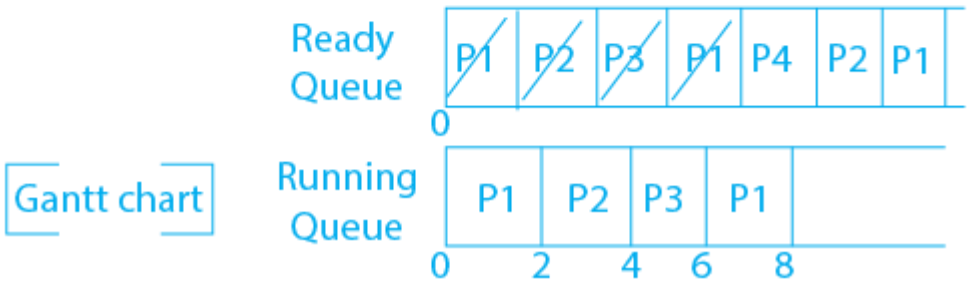
Process No	Arrived Time	Brust Time
P1	0	1
P2	1	2
P3	2	0
P4	4	1



In the above image, we can see that we have pushed process P1 from the ready queue to the running queue. We also decreased the burst time of the process P1 as it already executed 2 units.

Step 8: Now we will push all the processes arrival time within 8 whose burst time is not 0.

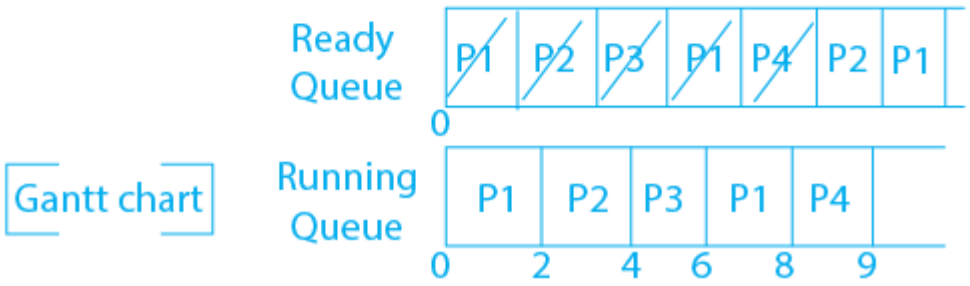
Process No	Arrived Time	Brust Time
P1	0	1
P2	1	2
P3	2	0
P4	4	1



In the above image, we can see that process P1 also has a remaining burst time so we will also push process P1 into the ready queue again.

Step 9: Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.

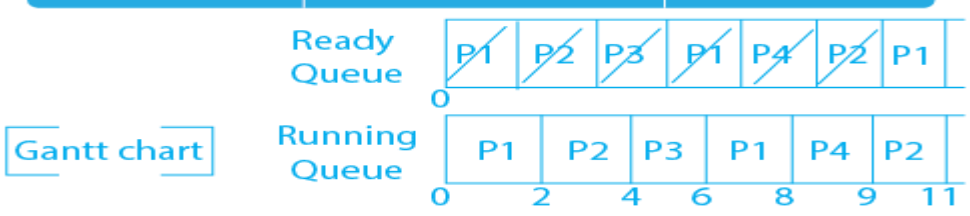
Process No	Arrived Time	Brust Time
P1	0	1
P2	1	2
P3	2	0
P4	4	0



In the above image, we can see that we have pushed process P4 from the ready queue to the running queue. We also decreased the burst time of the process P4 as it already executed 1 unit. Now, process P4's burst time becomes 0 so we will not consider it further.

Step 10: Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.

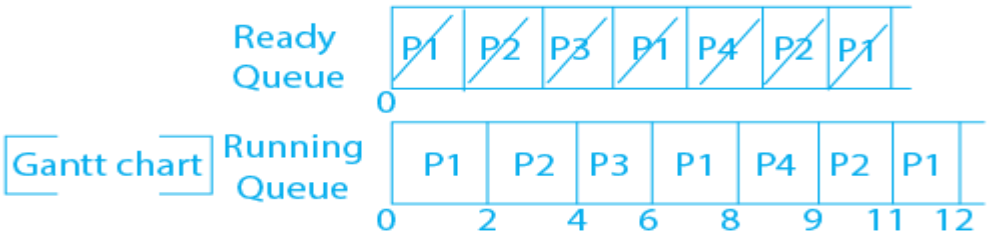
Process No	Arrived Time	Brust Time
P1	0	1
P2	1	0
P3	2	0
P4	4	0



In the above image, we can see that we have pushed process P2 from the ready queue to the running queue. We also decreased the burst time of the process P2 as it already executed 2 units. Now, process P2's burst time becomes 0 so we will not consider it further.

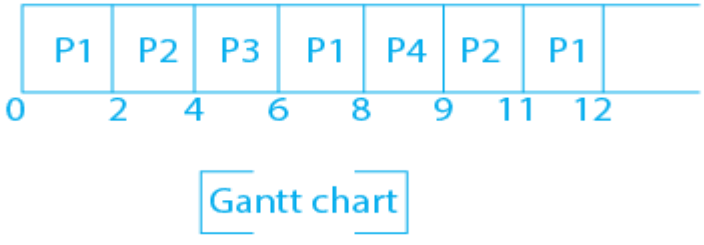
Step 11: Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.

Process No	Arrived Time	Brust Time
P1	0	0
P2	1	0
P3	2	0
P4	4	0



In the above image, we can see that we have pushed process P1 from the ready queue to the running queue. We also decreased the burst time of the process P1 as it already executed 1 unit. Now, process P1's burst time becomes 0 so we will not consider it further. Now our ready queue is empty so we will not perform any task now.

After performing all the operations, our running queue also known as the **Gantt chart** will look like the below.



Let's calculate the other terms like Completion time, Turn Around Time (TAT), Waiting Time (WT), and Response Time (RT). Below are the equations to calculate the above terms.

Turn Around Time = Completion Time – Arrival Time

Waiting Time = Turn Around Time – Burst Time

Response Time = CPU first time – Arrival Time

Let's calculate all the details for the above example.

Process No	Arrived Time	Brust Time	completion Time	TAT	WT	RT
P1	0	5	12	12	7	0
P2	1	4	11	10	6	1
P3	2	2	6	4	2	2
P4	4	1	9	5	4	4

Advantages-

- It gives the best performance in terms of average response time.
- It is best suited for time sharing system, client server architecture and interactive system.
- It doesn't face the issues of starvation or convoy effect.
- All the jobs get a fair allocation of CPU.
- It deals with all process without any priority

Disadvantages-

- It leads to starvation for processes with larger burst time as they have to repeat the cycle many times.
- Its performance heavily depends on time quantum.
- Lower time quantum results in higher the context switching overhead in the system and Higher time quantum makes it as FCFS
- Finding a correct time quantum is a quite difficult task in this system.
- Priorities can not be set for the processes.

Multilevel queue scheduling

Multilevel queue scheduling is used when processes in the ready queue can be divided into different classes where each class has its own scheduling needs.

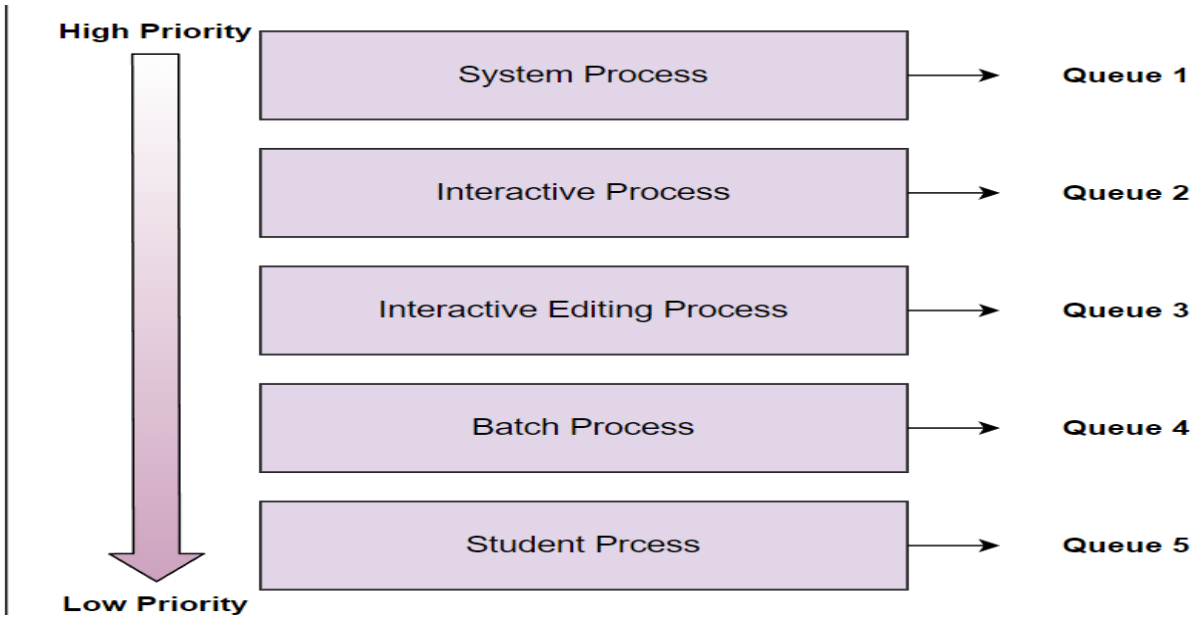
Example

Suppose we have five following processes of different nature. Let's schedule them using a multilevel queue scheduling:

Process ID	Process Type	Description
P1	System Process	The core process of OS.
P2	Interactive Process	A process with the some type of interaction.
P3	Interactive Editing Process	A type of interactive process.
P4	Batch Process	An OS feature that collects data and instructions into a batch before processing.
P5	Student Process	Process of lowest priority.

Solution

Let's handle this process scheduling using multilevel queue scheduling. To understand this better, look at the figure below:



The above illustration shows the division of queues based on priority. Every single queue has an absolute priority over the low-priority queue, and no other process is allowed to execute until the high-

priority queues are empty. For instance, if the interactive editing process enters the ready queue while the batch process is underway, then the batch process will be preempted

Advantages

1. You can use multilevel queue scheduling to apply different scheduling methods to distinct processes.
2. It will have low overhead in terms of scheduling.

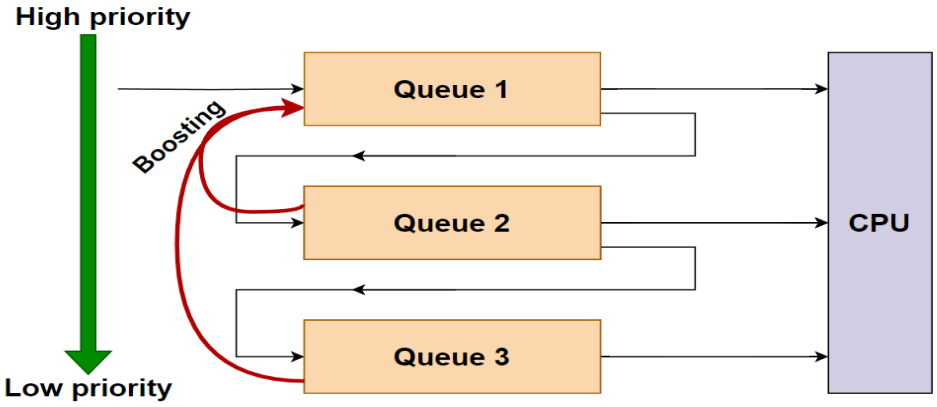
Disadvantages

1. There is a risk of starvation for lower priority processes.
2. It is rigid in nature.

Multilevel Feedback Queue Scheduling

Multilevel feedback queue scheduling it allows a process to move between the queue.

The diagram below illustrates the structure of an MLFQ. A new process is initially added to Queue 1, and if it fails to complete its execution, it moves to Queue 2 and vice versa. Once a process completes its execution, it is removed from the queue. Additionally, the MLFQ can utilize boosting to elevate low-level processes to higher queues.

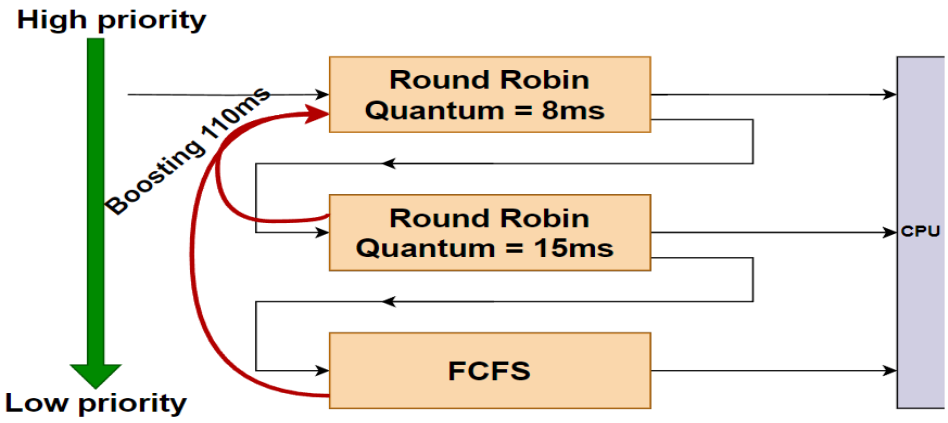


The steps involved in MLFQ scheduling when a process enters the system.

1. When a process enters the system, it is initially assigned to the highest priority queue.
2. The process can execute for a specific time quantum in its current queue.
3. If the process completes within the time quantum, it is removed from the system.
4. If the process does not complete within the time quantum, it is demoted to a lower priority queue and given a shorter time quantum.

5. This promotion and demotion process continues based on the behavior of the processes.
6. The high-priority queues take precedence over low-priority queues, allowing the latter processes to run only when high-priority queues are empty.
7. The feedback mechanism allows processes to move between queues based on their execution behavior.
8. The process continues until all processes are executed or terminated.

Example of MLFQ



There are various scheduling algorithms that can be applied to each queue, including FCFS, shortest remaining time, and round robin. Now, we will explore a multilevel feedback queue configuration consisting of three queues.

- Queue 1 (Q1) follows a round robin schedule with a time quantum of 8 milliseconds.
- Queue 2 (Q2) also uses a round robin schedule with a time quantum of 15 milliseconds.
- Finally, Queue 3 (Q3) utilizes a first come, first serve approach.
- Additionally, after 110 milliseconds, all processes will be boosted to Queue 1 for high-priority execution.

The multilevel feedback queue scheduler has the following parameters:

- The number of queues in the system.
- The scheduling algorithm for each queue in the system.
- The method used to determine when the process is upgraded to a higher-priority queue.
- The method used to determine when to demote a queue to a lower - priority queue.
- The method used to determine which process will enter in queue and when that process needs service.

Advantages of Multilevel Feedback Queue Scheduling:

- It is more flexible.
- It allows different processes to move between different queues.
- It prevents starvation by moving a process that waits too long for the lower priority queue to the higher priority queue.

Disadvantages of Multilevel Feedback Queue Scheduling:

- The selection of the best scheduler, it requires some other means to select the values.
- It produces more CPU overheads.
- It is the most complex algorithm.

Race Condition

When more than one process is executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition.

Example :

Suppose we have two process Producer and Consumer and counter variable is shared between these two. Initial Value of Counter is 5.

Producer Routine

register1 = counter

register1 = register1 + 1

counter= register1

Consumer Routine

register2 = counter

register2 = register2 - 1

counter= register2

Although both the producer and consumer routines above are correct separately, they may not function correctly when executed concurrently.

The concurrent execution of "counter++" and "counter--" is equivalent a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is To:

Consider this execution interleaving with "count = 5" initially:

S1: producer execute `register1 = counter`	{register1 = 5}
S2: producer execute `register1 = register1 + 1`	{register1 = 6}
S3: consumer execute `register2 = counter`	{register2 = 5}
S4: consumer execute `register2 = register2 - 1`	{register2 = 4}
S5: producer execute `counter = register1`	{counter = 6 }
S6: consumer execute `counter = register2`	{counter = 4}

Notice that we have arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state "counter== 6". We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

Critical Section Problem

The regions of a program that try to access shared resources and may cause race conditions are called critical section. To avoid race condition among the processes, we need to assure that only one process at a time can execute within the critical section.

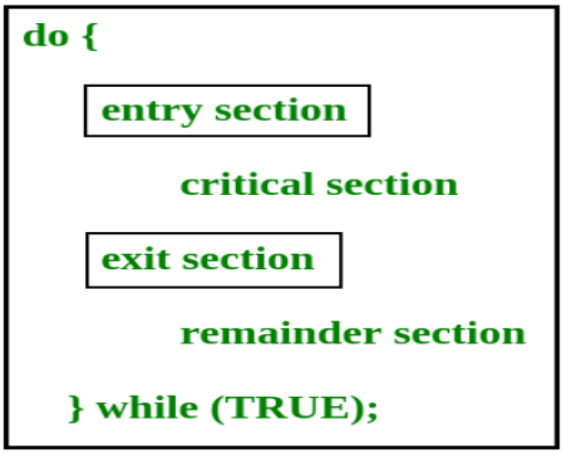
Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process**: The execution of one process does not affect the execution of other processes.
- **Cooperative Process**: A process that can affect or be affected by other processes executing in the system.

Process synchronization problem arises in the case of Cooperative processes also because resources are shared in Cooperative processes.

The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry Section The critical section may be followed by an exit Section. The remaining code is the remainder Section.

The general structure of a typical process Pi is shown in



Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress**: If no process is in the critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely. Essentially, the system must guarantee that every process can eventually enter its critical section.
- **Bounded Waiting**: There should be a **bound or a limit** on the number of times a particular **process can enter the critical section**. It should **not happen that the same process is taking up critical section every time** resulting in starvation of other processes. In this case, other processes would have to wait for an **infinite** amount of time, this should not happen.

Synchronization Hardware

Some times the problems of the Critical Section are also resolved by hardware. The hardware solution is as follows:

1. Disabling Interrupts
2. Test and Set
3. Swap

4. Unlock and Lock

Disabling Interrupts

The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by nonpreemptive kernels. Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

Test and Set

Test and set algorithm uses a boolean variable '**lock**' which is initially initialized to false. This lock variable determines the entry of the process inside the critical section of the code.

- The first process will enter the critical section at once as the value of lock is set to false by the initialization code.
- However, after the first process any process can go in. There is no queue

```
boolean lock = false;
boolean TestAndSet(boolean *target)
{
   boolean returnValue = *target;
   *target = true;
   return returnValue;
}
while(1)
{
while(TestAndSet(&lock));
   CRITICAL SECTION CODE;
lock = false;
```

REMAINDER SECTION CODE;

}

- In the above algorithm the TestAndSet () function takes a boolean value and returns the same value. TestAndSet () function sets the lock variable to true.
- When lock variable is initially false the TestAndSet(lock) condition checks for TestAndSet(false). As TestAndSet function returns the same value as its argument, TestAndSet(false) returns false. Now, while loop while (TestAndSet(lock)) breaks and the process enters the critical section.
- As one process is inside the critical section and lock value is now '**true**', if any other process tries to enter the critical section, then the new process checks for while (TestAndSet(true)) which will return **true** inside while loop and as a result the other process keeps executing the while loop.

- As no queue is maintained for the processes stuck in the while loop, bounded waiting is not ensured.

Swap

Swap function uses two boolean variables lock and key. Both lock and key variables are initially initialized to false

```
boolean lock = false;
individual key = false;
void swap(boolean *a, boolean  *b)
{
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
while(1)
{
```

```
  key=true;
  while(key){
     swap(&lock,&key);
  }
    CRITICAL SECTION CODE
  lock = false;
  REMAINDER SECTION CODE
}
```

- In the code above when a process P1 enters the critical section of the program it first executes the while loop. As key value is set to true just before the for loop so swap(lock, key) swaps the value of lock and key. Lock becomes true and the key becomes false. In the next iteration of the while loop breaks and the process, P1 enters the critical section.
- The value of lock and key when P1 enters the critical section is lock = true and key = false.
- Let's say another process, P2, tries to enter the critical section while P1 is in the critical section. Let's take a look at what happens if P2 tries to enter the critical section.
- **key is set to true** again after the first while loop is executed i.e while(1). Now, the second while loop in the program i.e while(key) is checked. As key is true the process enters the second while loop. swap(lock, key) is executed again. as both key and lock are true so after swapping also both will be true. So, the while keeps executing and the process P2 keeps running the while loop until Process P1 comes out of the critical section and makes **lock false**.
- When Process P1 comes out of the critical section the value of lock is again set to false so that other processes can now enter the critical section.
- When a process is inside the critical section than all other incoming process trying to enter the critical section is not maintained in any order or queue. So any process out of all the waiting process can get the chance to enter the critical section as the lock becomes false. So, there may be a process that may wait indefinitely. So, **bounded waiting is not ensured in Swap algorithm also**.

Unlock and lock

- Unlock and Lock Algorithm uses TestAndSet to regulate the value of lock but it adds another value, waiting[i], for each process which checks whether or not a process has been waiting.
- A ready queue is maintained with respect to the process in the critical section. All the processes coming in next are added to the ready queue with respect to their process number, not necessarily sequentially.
- Once the ith process gets out of the critical section, it does not turn lock to false so that any process can avail the critical section now, which was the problem with the previous algorithms. Instead, it checks if there is any process waiting in the queue.
- The queue is taken to be a circular queue. j is considered to be the next process in line and the while loop checks from jth process to the last process and again from 0 to (i-1)th process if there is any process waiting to access the critical section.
- If there is no process waiting then the lock value is changed to false and any process which comes next can enter the critical section.
- If there is, then that process' waiting value is turned to false, so that the first while loop becomes false and it can enter the critical section. This ensures bounded waiting. So the problem of process synchronization can be solved through this algorithm.

```
// Shared variable lock initialized to false
// and individual key initialized to false
boolean lock = false;
boolean key = false;
boolean waiting[i];
boolean TestAndSet(boolean *target)
{
   boolean returnValue = *target;
  * target = true;
```

```
   return returnValue;
}

while(1){
   waiting[i] = true;
   key = true;
   while(waiting[i] && key){
      key = TestAndSet(&lock);
   }
   CRITICAL SECTION CODE
   j = (i+1) % n;
   while(j != i && !waiting[j])
      j = (j+1) % n;
   if(j == i)
      lock = false;
   else
      waiting[j] = false;
   REMAINDER SECTION CODE
}
```

Semaphores

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple non negative integer variable to synchronize the progress of interacting processes. This non negative integer variable is called a **semaphore .** So, it is basically a synchronizing tool and is accessed only through two low standard atomic operations, **wait** and **signal** designated by P(S) and V(S) respectively.

Entry to the critical section is controlled by the wait operation and exit from a critical region is taken care by signal operation.

Wait Operation (P):

```
Wait(S)
{
while (S<=0);// no operation
S--;
}
```

- When a process/thread wants to access a shared resource, it first performs a "wait" operation on the semaphore associated with that resource.
- If the semaphore value is positive (greater than 0), it decrements the value by 1 and continues accessing the resource.
- If the semaphore value is zero, indicating that the resource is currently being used by another process/thread, the process/thread is blocked or put to sleep until the semaphore becomes available.

Signal Operation (V):

```
Signal(S)
{

   S++;
}
```

- When a process/thread finishes using a shared resource, it performs a "signal" operation on the semaphore, which increments the semaphore value by 1.

- This operation indicates that the resource is now available for other processes/threads waiting on it. If there are any blocked processes/threads waiting for the semaphore, one of them is awakened and granted access to the resource.

Types of Semaphores-

There are mainly two types of semaphores-

1. Counting Semaphores
2. Binary Semaphores or Mutexes

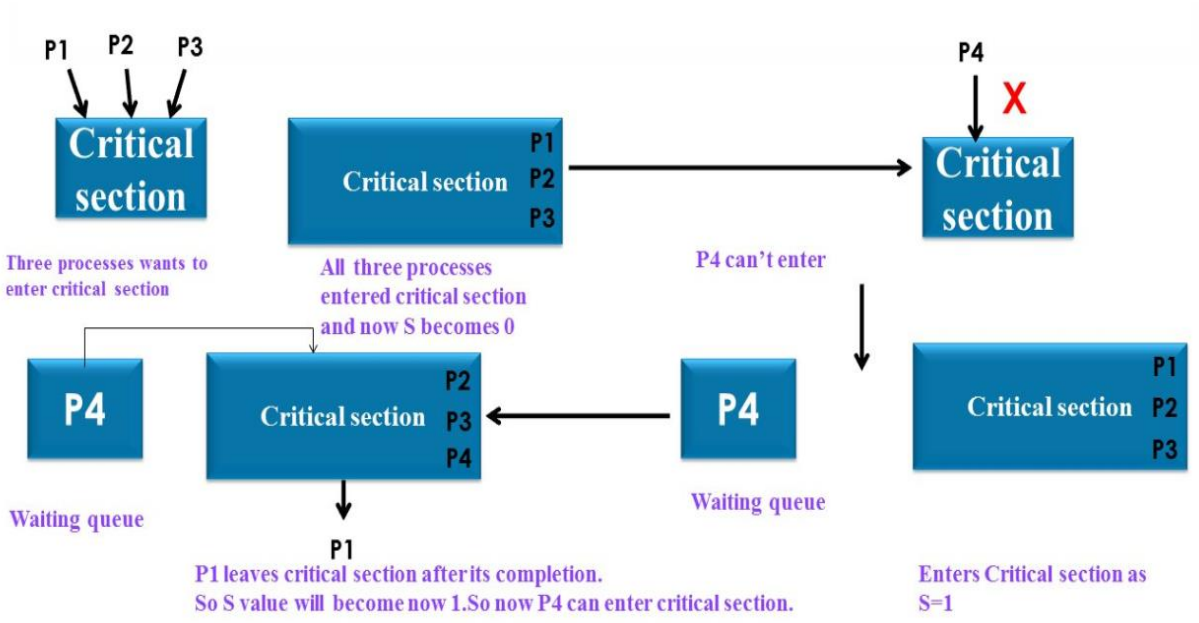
```
do {
wait (S) ;
// critical section
signal(S);
//remainder section
} while (TRUE)
```

Counting Semaphores

1. Initialize the counting semaphore with a value that represents the maximum number of resources that can be accessed simultaneously.
2. When a process attempts to access the shared resource, it first attempts to acquire the semaphore using the `wait()` or `P()` function.
3. The semaphore value is checked. If it is greater than zero, the process is allowed to proceed and the value of the semaphore is decremented by one. If it is zero, the process is blocked and added to a queue of waiting processes.
4. When a process finishes accessing the shared resource, it releases the semaphore using the `signal()` or `V()` function.
5. The value of the semaphore is incremented by one, and any waiting processes are unblocked and allowed to proceed.
6. Multiple processes can access the shared resource simultaneously as long as the value of the semaphore is greater than zero.

7. The counting semaphore provides a way to manage access to shared resources and ensure that conflicts are avoided, while also allowing multiple processes to access the resource at the same time.

- Assuming Initial Value of Semaphore=3

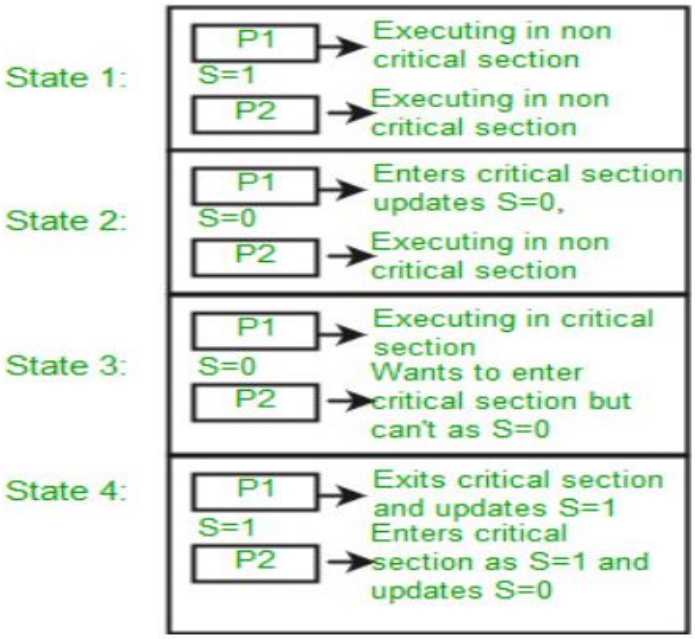


Binary Semaphore

This is also known as a mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.

Implementation of Binary Semaphore

- Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1.
- Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0.
- Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s.
- This way mutual exclusion is achieved. Look at the below image for details which is a Binary semaphore.



Criteria	Binary Semaphore	Counting Semaphore
Definition	A Binary Semaphore is a semaphore whose integer value range over 0 and 1.	A counting semaphore is a semaphore that has multiple values of the counter. The value can range over an unrestricted domain.
Structure Implementation	typedef struct { int semaphore_variable; } binary_semaphore;	typedef struct { int semaphore_variable; Queue list; //A queue to store the list of task }counting_semaphore;
Representation	0 means that a process or a thread is accessing the critical section, other process should wait for it to exit the critical section. 1 represents the critical section is free.	The value can range from 0 to N, where N is the number of process or thread that has to enter the critical section.
Mutual Exclusion	Yes, it guarantees mutual exclusion, since just one process or thread can enter the critical section at a time.	No, it doesn't guarantees mutual exclusion, since more than one process or thread can enter the critical section at a time.

Bounded wait	No, it doesn't guarantees bounded wait, as only one process can enter the critical section, and there is no limit on how long the process can exist in the critical section, making another process to starve.	Yes, it guarantees bounded wait, since it maintains a list of all the process or threads, using a queue, and each process or thread get a chance to enter the critical section once. So no question of starvation.
Starvation	No waiting queue is present then FCFS (first come first serve) is not followed so,starvation is possible and busy wait present	Waiting queue is present then FCFS (first come first serve) is followed so,no starvation hence no busy wait.
Number of instance	Used only for a single instance of resource type R.it can be usedonly for 2 processes.	Used for any number of instance of resource of type R.it can be used for any number of processes.

- The main disadvantage of the semaphore is that it requires busy waiting. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock because the process spins while waiting for the lock.
- To overcome the need for busy waiting, we can modify the definition of wait () and Signal () semaphore operations. When the process executes the wait () operation and finds that the semaphore value is not positive it must wait. Rather that engaging in busy waiting the process can block itself.
- The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to the waiting state. The process that is blocked waiting on a semaphore S should be restarted when some other process executes a signal () operation, the process is restarted by a wakeup () operation.

Definition of semaphore as a C

struct typedef struct

```
{
int value;
 struct process *list;
} semaphore;
```

- Each semaphore has an integer value and a list of processes list.
- When a process must wait on a semaphore, it is added to the list of processes.
- A signal() operation remove one process from the list of waiting processes and awakens that process.

Wait () operation can be defined as

```
Wait (semaphore *s)
{
S->value--;
If (S->Value<0)
{
Add this Process to s->value list;
block ();
} }
```

Signal operation can be defined as

```
 Signal (semaphore *S)
{
 S->value++;
If (S-> value <=0)
{
Remove a process P from S-> list;
Wakeup (P);
} }
```

The block () operation suspends the process that invokes it. The wakeup () operation resumes the execution of a blocked process P.

Deadlock and Starvation:

The implementation of semaphore with a waiting queue may result in a situation where two more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. When such a state is reached that process are said to be deadlocked.

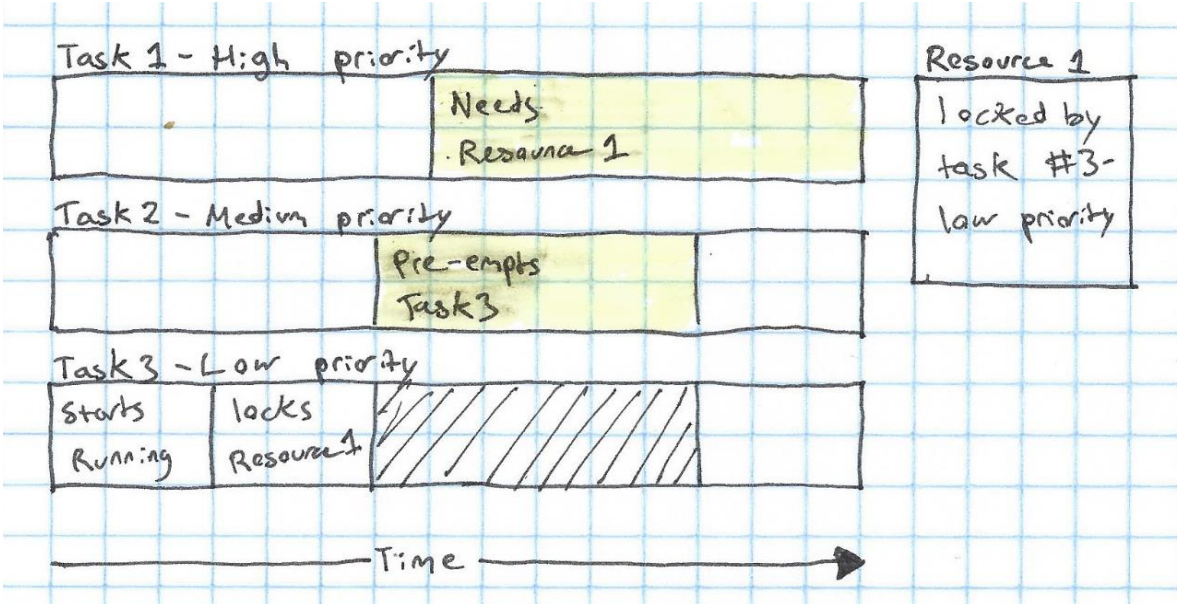
P0	P1
Wait(S);	wait(Q);
Wait(Q);	Wait(S);
. . . . .	
Signal(S);	Signal (Q);
Signal (Q);	Signal (S);

P0 executes wait (S) and then P1 executes wait (Q). When p0 executes wait (Q), it must wait until p1 executes Signal (Q) in the same way P1 must wait until P0 executes signal (S). So p0 and p1 are deadlocked. The other problem related to deadlocks is indefinite blocking or starvation, a situation where processes wait indefinitely within the semaphore

P0 executes wait (S) and then P1 executes wait (Q). When p0 executes wait (Q), it must wait until p1 executes Signal (Q) in the same way P1 must wait until P0 executes signal (S). So p0 and p1 are deadlocked. The other problem related to deadlocks is indefinite blocking or starvation, a situation where processes wait indefinitely within the semaphore.

Priority inversion

Priority inversion is a phenomenon that can occur in an operating system where a higher-priority task is blocked because it is waiting for a lower-priority task to release a resource it needs. This can happen in a system where tasks or threads have different priorities, and a lower-priority task is currently holding a resource that a higher-priority task needs to execute.



For example, consider a system with three tasks: 1, 2, and 3. Task 1 has the highest priority, task 2 has medium priority, and task 3 has the lowest priority. If task 3 is currently in Critical Section and task 1 needs to enter into critical section in that case task 1 may be blocked until task 2 completes, even though task 1 has a higher priority than task2.This is known as Bounded Priority Inversion.

In meantime Task2 just wants a normal execution and doesnot want to enter into critical section.Then according to priority scheduling task 2 can pre-empt task 1 (but still task 1 holds the lock of Critical section) and continue the normal execution.Now task1 has to wait for completion of task 2 and task1 though the priorities of the both are lower than the Task1.This is known as Unbounded Priority Inversion.

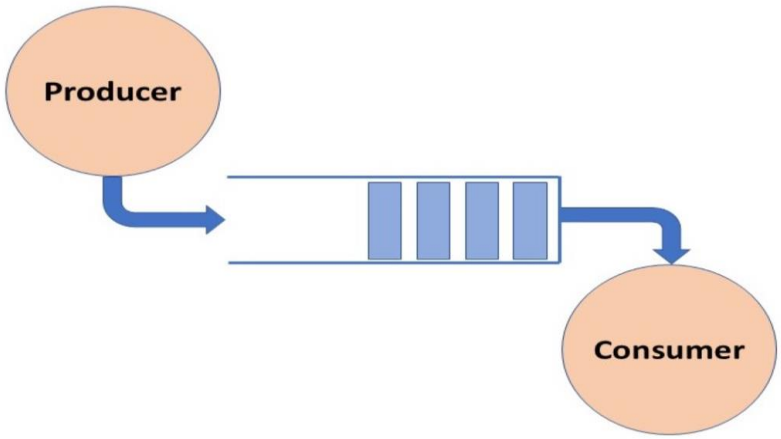
Priority inversion can cause significant problems in real-time systems, where the timely execution of high-priority tasks is critical. If a high-priority task is blocked for an extended period due to priority inversion, it can cause delays and potentially impact the system's overall performance.

One of the solutions to this problem is *Priority Inheritance*. In *Priority Inheritance*, when L is in the critical section, L inherits the priority of H at the time when H starts pending for the critical section. By doing so, M doesn't interrupt L and H doesn't wait for M to finish. Please note that inheriting of priority is done temporarily i.e. L goes back to its old priority when L comes out of the critical section.

Classical Problems of Synchronization

Bounded Buffer Problem(Producer - consumer problem) is example classic problems of synchronization. Producer process produce data item that consumer process consumes later.

• Buffer is used between producer and consumer. Buffer size may be fixed or variable. The producer portion of the application generates data and stores it in a buffer, and the consumer reads data from the buffer.



- The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
- Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
- Accessing memory buffer should not be allowed to producer and consumer at the same time.

The solution to the Producer-Consumer problem involves three *semaphore* variables.

- **semaphore Full**: Tracks the space filled by the Producer process. It is initialized with a value of 0 as the buffer will have 0 filled spaces at the beginning
- **semaphore Empty**: Tracks the empty space in the buffer. It is initially set to **buffer_size** as the whole buffer is empty at the beginning.
- **semaphore mutex**: Used for mutual exclusion so that only one process can access the shared buffer at a time..

At any particular time, the current value of empty denotes the number of vacant slots in the buffer, while full denotes the number of occupied slots.

```
do
{
   // process will wait until the empty > 0 and further decrement of 'empty'
   wait(empty);
   // To acquire the lock
   wait(mutex);

   /* Here we will perform the insert operation in a particular slot */

   // To release the lock
   signal(mutex);
   // increment of 'full'
   signal(full);
}
while(TRUE)
```

- When we look at the above code for a producer, we can see that it first waits until at least one slot is vacant.
- wait(empty) decreases the value of the semaphore variable "empty" by one, indicating that when the producer produces anything, the value of the empty space in the buffer decreases. If the buffer is full, or the value of the semaphore variable "empty" is 0, the program will stop and no production will take place.
- wait(mutex) sets the semaphore variable "mutex" to zero, preventing any other process from entering the critical section.
- The buffer is then locked, preventing the consumer from accessing it until the producer completes its function.
- signal(mutex) is being used to mark the semaphore variable "mutex" to "1" so that other processes can arrive into the critical section though because the production is finished and the insert operation is also done.
- So, After the producer has filled a slot in the buffer, the lock is released.

- signal(full) is utilized to increase the semaphore variable "full" by one because after inserting the data into the buffer, one slot is filled in the buffer and the variable "full" must be updated.

```
do
{
   // need to wait until full > 0 and then decrement the 'full'
   wait(full);
   // To acquire the lock
   wait(mutex);

    /* Here we will perform the remove operation in a particular slot */

   // To release the lock
   signal(mutex);
   // increment of 'empty'
   signal(empty);
}
while(TRUE);
```

- The consumer waits until the buffer has at least one full slot.
- wait(full) is used to reduce the semaphore variable "full" by one since the variable "full" must be reduced by one of the consumers consuming some data.
- wait(mutex) sets the semaphore variable "mutex" to "0", preventing any other processes from entering the critical section.
- And soon after that, the consumer then acquires a lock on the buffer.
- The consumer then completes the data removal operation by removing data from one of the filled slots.
- So because the consumption and remove operations are complete, signal(mutex) is being used to set the semaphore variable "mutex" to "1" so that other processes can enter the critical section now.
- The lock is then released by the consumer.

- Because one slot space in the buffer is released after extracting the data from the buffer, signal(empty) is used to raise the variable "empty" by one.

Reader's writer problem

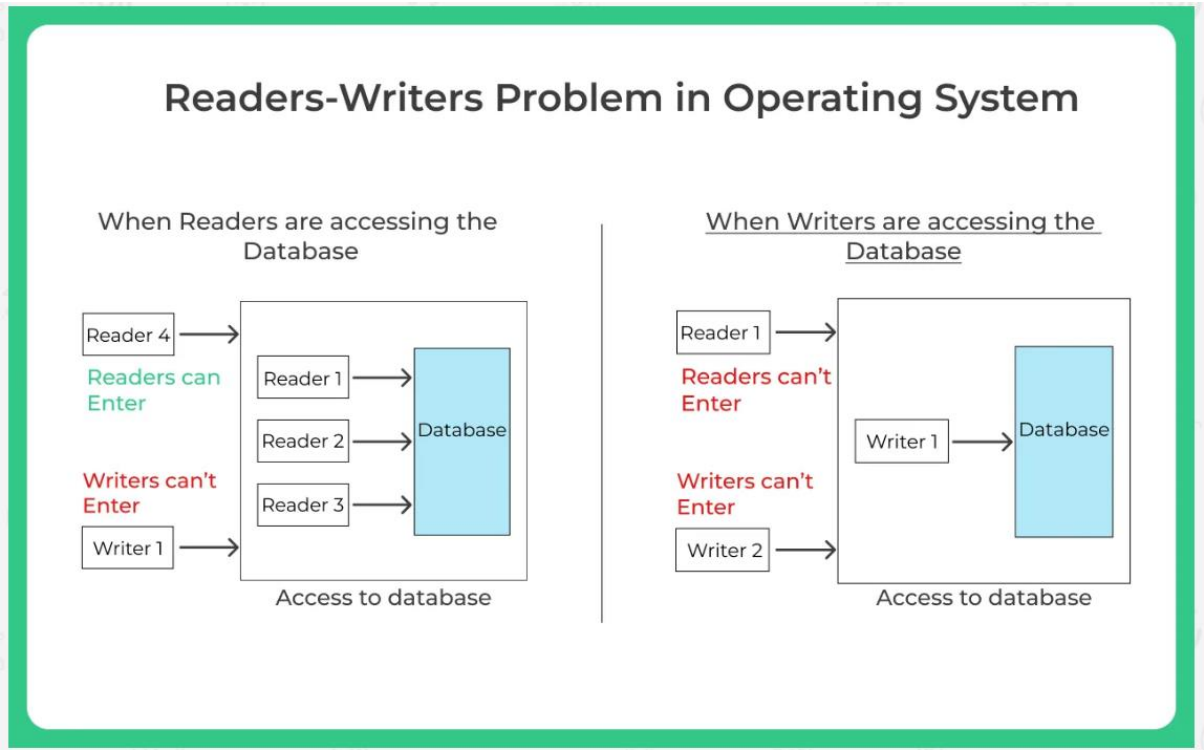
The readers-writers problem relates to an object such as a file that is shared between multiple processes. . The problem statement is, if a database or file is to be shared among several concurrent process, there can be broadly 2 types of users

- Readers – Reader are those processes/users which only read the data
- Writers – Writers are those processes which also write, that is, they change the data .

It is allowed for 2 or more readers to access shared data, simultaneously as they are not making any change and even after the reading the file format remains the same.

But if one writer(Say w1) is editing or writing the file then it should locked and no other writer(Say w2) can make any changes until w1 has finished writing the file.

Writers are given to be exclusive access to shared database while writing to database. This is called Reader's writer problem.



Variables used –

- Mutex – mutex (used for mutual exclusion, when readcount is changed) initialised as 1
- Semaphore – wrt (used by both readers and writers) initialised as 1
- readers_count – Counter of number of people reading the filei initialised as 0

Functions –

There are two functions –

1. wait() – performs as –, which basically decrements value of semaphore
2. signal() – performs as ++. which basically increments value of semaphore

Reader Process

```
while (TRUE)
{
   // Acquire lock
   wait(m);
   readCount++;
   if (readCount == 1)
{
      wait(w);
   }

   // Release lock
   signal(m);

   wait(m);
readCount--;
   if (readCount == 0)
{
      signal(w);
   }

   // Release lock
   signal(m);
}
```

Writer Process

```
do {
```

```
wait(wrt);
// writing is performed
signal(wrt);
} while (TRUE);
```

Dining philosopher's problem

The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available.

Solution of Dining Philosophers Problem

A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

The structure of the chopstick is shown below −

```
semaphore chopstick [5];
```

Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.

The structure of a random philosopher i is given as follows −

```
do {
  wait( chopstick[i] );
  wait( chopstick[ (i+1) % 5] );
  . .
  . EATING THE RICE
  .
  signal( chopstick[i] );
  signal( chopstick[ (i+1) % 5] );
  .
  . THINKING
  .
} while(1);
```

In the above structure, first wait operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

Difficulty with the solution

The above solution makes sure that no two neighboring philosophers can eat at the same time. But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are as follows −

- There should be at most four philosophers on the table.

- An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
- A philosopher should only be allowed to pick their chopstick if both are available at the same time.

Monitors

Semaphores can be very useful for solving concurrency problems**, but only if programmers use them properly**. If even one process fails to abide by the proper use of semaphores, either accidentally or deliberately, then the whole system breaks down.

- **Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:**

signal(mutex);
critical section
wait(mutex);

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- **Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes**

wait(mutex);
critical section
wait(mutex);

In this case, a deadlock will occur.

- **Suppose that a process omits the wait (mutex), or the signal (mutex), or both.**

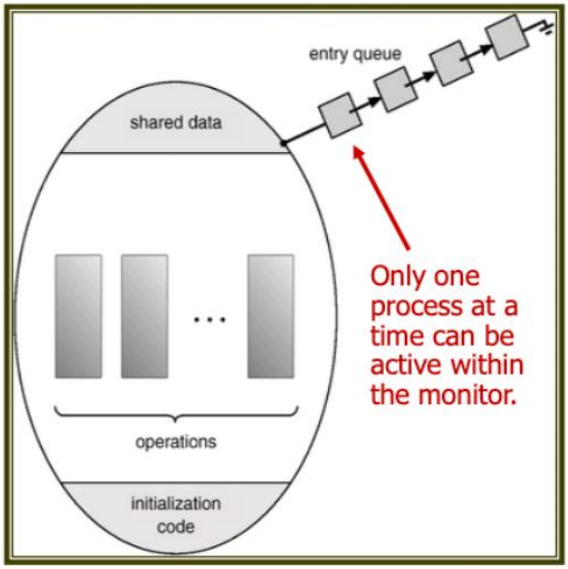
In this case, either mutual exclusion is violated or a deadlock will occur.

For this reason a higher-level language construct has been developed, called monitors.

Monitor Usage

A monitor is essentially a class, in which all data is private, and with the special restriction that only one method within any given monitor object may be active at the same time. An additional restriction is that monitor methods may only access the shared data within the monitor and any data passed to them as parameters. I.e. they cannot access any data external to the monitor.

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
            . . .
    }
    procedure body Pn (...) {
            . . .
    }
    {
            initialization code
    }
}
```



In order to fully realize the potential of monitors, we need to introduce one additional new data type, known as a **condition.**A variable of type condition has only two legal operations, wait and signal. I.e. if X was defined as type condition, then legal operations would be X.wait( ) and X.signal( ).

The wait operation blocks a process until some other process calls signal, and adds the blocked process onto a list associated with that condition.

The signal process does nothing if there are no processes waiting on that condition. Otherwise it wakes up exactly one process from the condition's list of waiting processes.

Signal and wait - When process P issues the signal to wake up process Q, P then waits, either for Q to leave the monitor or on some other condition.

Signal and continue - When P issues the signal, Q waits, either for P to exit the monitor or for some other condition.

Dining Philosophers Solution using Monitors

Monitors are used because they give a deadlock free solution to the Dining Philosophers problem. It is used to gain access over all the state variables and condition variables. After implying monitors, it imposes a restriction that a philosopher may pickup his chopsticks only if both of them are available at the same time.

To code the solution, we need to distinguish among three states in which may find a philosopher.

- THINKING
- HUNGRY
- EATING

Example

Here is implementation of the Dining Philosophers problem using Monitors –

```
monitor DiningPhilosophers {
  enum {THINKING, HUNGRY, EATING} state[5];
  condition self[5];
  void pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING) {
      self[i].wait();
    }
  }
  void putdown(int i) {
    state[i] = THINKING;
    test((i + 4) % 5);
    test((i + 1) % 5);
  }
  void test(int i) {
    if (state[(i + 4) % 5] != EATING &&
    state[i] == HUNGRY &&
    state[(i + 1) % 5] != EATING) {
      state[i] = EATING;
      self[i].signal();
    }
```

```
  }
  initialization code() {
    for(int i=0;i<5;i++)
    state[i] = THINKING;
  }
}
```

Before start eating, each philosopher must invoke the pickup() operation. It indicates that the philosopher is hungry, means that the process wants to use the resource. It also set the state to EATING in test() only if the philosopher's left and right neighbors are not eating. If the philosopher is unable to eat, then wait() operation is invoked. After the successful completion of the operation, the philosopher may now eat.

Keeping that in mind, the philosopher now invokes the putdown() operation. After leaving forks, it checks on his neighbors. If they are HUNGRY and both of its neighbors are not EATING, then invoke signal() and offer them to eat.

Thus a philosopher must invokes the pickup() and putdown() operations simultaneously which ensures that no two neighbors are eating at the same time, thus achieving mutual exclusion. Thus, it prevents the deadlock. But there is a possibility that one of the philosopher may starve to death.

Implementing Monitor using Semaphore

Let's implement a a monitor mechanism using semaphores.

Following is a step-by-step implementation:

Step 1: Initialize a semaphore mutex to 1.

Step 2: Provide a semaphore mutex for each monitor.

Step 3: A process must execute wait (mutex) before entering the monitor and must execute signal (mutex) after leaving the monitor.

Step 4: Since a signaling process must wait until the resumed process either leaves or waits, introduce an additional semaphore, S, and initialize it to 0.

Step 5: The signaling processes can use S to suspend themselves. An integer variable S_count is also provided to count the number of processes suspended next. Thus, each external function Fun is replaced by

wait (mutex);

body of Fun

if (S_count > 0)

signal (S);

else

signal (mutex);

Mutual exclusion within a monitor is ensured.

Let's see how condition variables are implemented.

Step 1: x is condition.

Step 2: Introduce a semaphore x_num and an integer variable x_count.

Step 3: Initialize both semaphores to 0.

x.wait() is now implemented as: x.wait()

x_count++;

if (S_count > 0)

signal (S);

else

signal (mutex);

wait (x_num);

x_count–;

The operation x.signal () can be implemented as

if (x_count > 0) {

S_count++;

signal (x_num);

wait (S);

S_count–;}

Main Differences between the Semaphore and Monitor

Some of the main differences are as follows:

1. A semaphore is an integer variable that allows many processes in a parallel system to manage access to a common resource like a multitasking OS. On the other hand, a monitor is a synchronization technique that enables threads to mutual exclusion and the **wait()** for a given condition to become true.
2. When a process uses shared resources in semaphore, it calls the **wait()** method and blocks the resources. When it wants to release the resources, it executes the **signal()** In contrast, when a process uses shared resources in the monitor, it has to access them via procedures.
3. Semaphore is an integer variable, whereas monitor is an abstract data type.
4. In semaphore, an integer variable shows the number of resources available in the system. In contrast, a monitor is an abstract data type that permits only a process to execute in the crucial section at a time.
5. Semaphores have no concept of condition variables, while monitor has condition variables.
6. A semaphore's value can only be changed using the **wait()** and **signal()** In contrast, the monitor has the shared variables and the tool that enables the processes to access them.