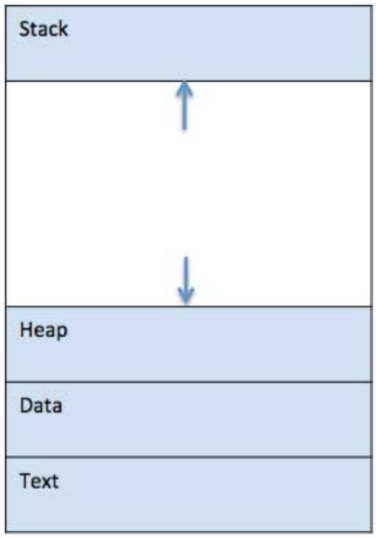
Unit-3

Process Management

Process

A process is basically a program in execution. The execution of a process must progress in a sequential fashion. 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.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data.



Stack

The process Stack contains the temporary data such as method/function parameters, return address and local variables.

Heap

This is dynamically allocated memory to a process during its run time.

Text

This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.

Data

This section contains the global and static variables.

The Process Model

In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of **sequential processes**, or just processes for short. A process is just an executing program, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel, than to try to keep track of how the CPU switches from program to program. This rapid switching back and forth is called **multiprogramming.**

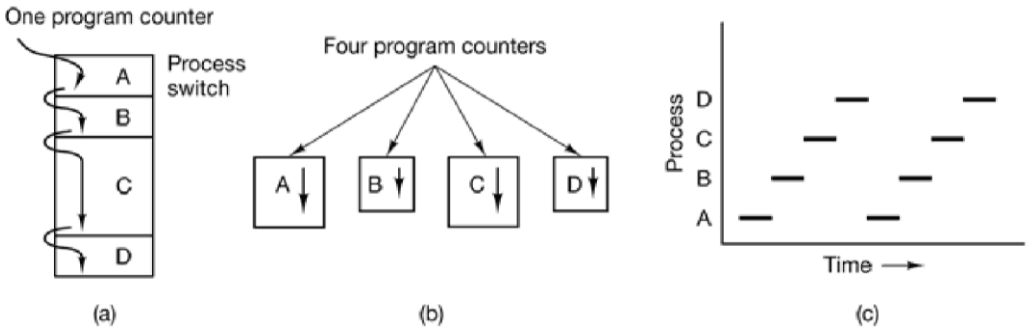


Fig: (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform and probably not even reproducible if the same processes are run again. Thus, processes must not be programmed with built-in assumptions about timing. The difference between a process and a program is subtle, but crucial. An analogy makes help here:

Consider a culinary-minded computer scientist who is baking a birthday cake for his daughter. He has a birthday cake recipe and a kitchen well stocked with all the input: flour, eggs, sugar, extract of vanilla, and so on. In this analogy, the recipe is the program (i.e., an algorithm expressed in some suitable notation), the computer scientist is the processor (CPU), and the cake ingredients are the input data. The process is the activity consisting of our baker reading the recipe, fetching the ingredients, and baking the cake.

Now imagine that the computer scientist's son comes running in crying, saying that he has been stung by a bee. The computer scientist records where he was in the recipe (the state of the current process is saved), gets out a first aid book, and begins following the directions in it. Here we see the processor being switched from one process (baking) to a higher-priority process (administering medical care), each having a different program (recipe versus first aid book). When the bee sting has been taken care of, the computer scientist goes back to his cake, continuing at the point where he left off.

The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state. A single processor may be shared among several processes, with some scheduling algorithm being used to determine when to stop work on one process and service a different one.

Process States and Transitions

The process, from its creation to completion, passes through various states. Although each process is an independent entity, with its own program counter and internal state, processes often need to interact with other processes. One process may generate some output that another process uses as input. In the shell command

```
cat chapter1 chapter2 chapter3 | grep tree
```

the first process, running *cat*, concatenates and outputs three files. The second process, running *grep*, selects all lines containing the word ‘‘tree.’’ Depending on the relative speeds of the two processes (which depends on both the relative complexity of the programs and how much CPU time each one has had), it may happen that *grep* is ready to run, but there is no input waiting for it. It must then block until some input is available.

When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. It is also possible for a process that is conceptually ready and able to run to be stopped because the operating system has decided to allocate the CPU to another process for a while. These two conditions are completely different. In the first case, the suspension is inherent in the problem (you cannot process the user’s command line until it has been typed). In the second case, it is a technicality of the system (not enough CPUs to give each process its own private processor).

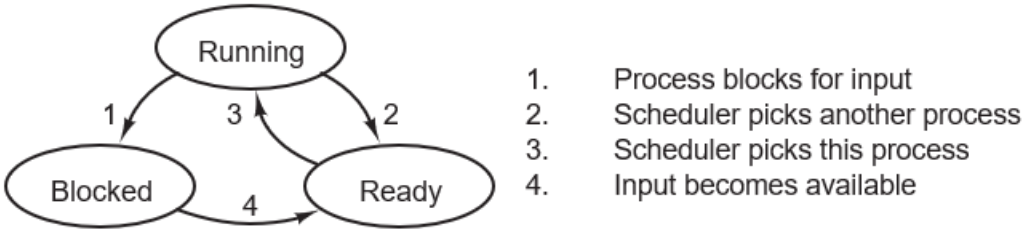


Fig: Process states and Transitions

1. Running (actually using the CPU at that instant).
2. Ready (runnable; temporarily stopped to let another process run).
3. Blocked (unable to run until some external event happens).

Logically, the first two states are similar. In both cases the process is willing to run, only in the second one, there is temporarily no CPU available for it. The third state is fundamentally different from the first two in that the process cannot run, even if the CPU is idle and has nothing else to do.

Four **transitions** are possible among these three states, as shown. **Transition 1** occurs when the operating system discovers that a process cannot continue right now. In some systems the process can execute a system call, such as pause, to get into blocked state. In other systems, including UNIX, when a process reads from a pipe or special file (e.g., a terminal) and there is no input available, the process is automatically blocked.

Transitions 2 and 3 are caused by the process scheduler, a part of the operating system, without the process even knowing about them. **Transition 2** occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. **Transition 3** occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again. The subject of scheduling, that is, deciding which process should run when and for how long, is an important one; we will look at it later in this chapter. Many algorithms have been devised to try to balance the competing demands of efficiency for the system as a whole and fairness to individual processes.

Transition 4 occurs when the external event for which a process was waiting (such as the arrival of some input) happens. If no other process is running at that instant, transition 3 will be triggered and the process will start running. Otherwise it may have to wait in *ready* state for a little while until the CPU is available and its turn comes.

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.

It is very important for process management as the data structuring for processes is done in terms of the PCB. It also defines the current state of the operating system.

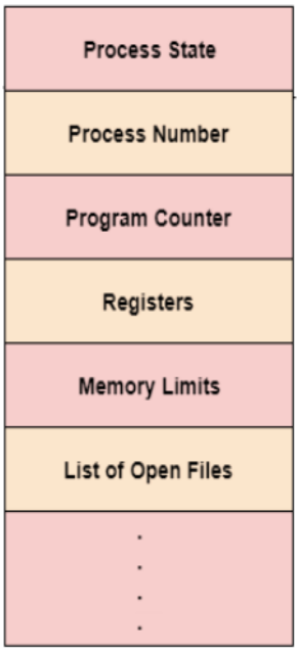


Fig: Process Control Block

Process State

This specifies the process state i.e. new, ready, running, waiting or terminated.

Process Number

This shows the number of the particular process.

Program Counter

This contains the address of the next instruction that needs to be executed in the process.

Registers

This specifies the registers that are used by the process. They may include accumulators, index registers, stack pointers, general purpose registers etc.

List of Open Files

These are the different files that are associated with the process

CPU Scheduling Information

The process priority, pointers to scheduling queues etc. is the CPU scheduling information that is contained in the PCB. This may also include any other scheduling parameters.

Memory Management Information

The memory management information includes the page tables or the segment tables depending on the memory system used. It also contains the value of the base registers, limit registers etc.

I/O Status Information

This information includes the list of I/O devices used by the process, the list of files etc.

Accounting information

The time limits, account numbers, amount of CPU used, process numbers etc. are all a part of the PCB accounting information.

Location of the Process Control Block

The process control block is kept in a memory area that is protected from the normal user access. This is done because it contains important process information. Some of the operating systems place the PCB at the beginning of the kernel stack for the process as it is a safe location.

Operations on the Process

The execution of a process is a complex activity. It involves various operations. Following are the operations that are performed while execution of a process:

1. **Creation:** This the initial step of process execution activity. Process creation means the construction of a new process for the execution. This might be performed by system, user or old process itself. There are several events that leads to the process creation. Some of the such events are following:

    - When we start the computer, system creates several background processes.
    - A user may request to create a new process.
    - A process can create a new process itself while executing.
    - Batch system takes initiation of a batch job.

2. Scheduling/Dispatching: The event or activity in which the state of the process is changed from ready to running. It means the operating system puts the process from ready state into the running state. Dispatching is done by operating system when the resources are free or the process has higher priority than the ongoing process. There are various other cases in which the process in running state is preempted and process in ready state is dispatched by the operating system.

3. Blocking: When a process invokes an input-output system call that blocks the process and operating system put in block mode. Block mode is basically a mode where process waits for input-output. Hence on the demand of process itself, operating system blocks the process and dispatches another process to the processor. Hence, in process blocking operation, the operating system puts the process in 'waiting' state.

4. Preemption: When a timeout occurs that means the process hadn't been terminated in the allotted time interval and next process is ready to execute, then the operating system preempts the process. This operation is only valid where CPU scheduling supports preemption. Basically this happens in priority scheduling where on the incoming of high priority process the ongoing process is preempted. Hence, in process preemption operation, the operating system puts the process in 'ready' state.

5. Termination: Process termination is the activity of ending the process. In other words, process termination is the relaxation of computer resources taken by the process for the execution. Like creation, in termination also there may be several events that may lead to the process termination. Some of them are:

- Process completes its execution fully and it indicates to the OS that it has finished.
- Operating system itself terminates the process due to service errors.
- There may be problem in hardware that terminates the process.
- One process can be terminated by another process.

Process Hierarchies

In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a process hierarchy. Note that unlike plants and animals that use sexual reproduction, a process has only one parent (but zero, one, two, or more children). So a process is more like a hydra than like, say, a cow.

In UNIX, a process and all of its children and further descendants together form a process group. When a user sends a signal from the keyboard, the signal is delivered to all members of the process group currently associated with the keyboard (usually all active processes that were created in the current window). Individually, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

As another example of where the process hierarchy plays a key role, let us look at how UNIX initializes itself when it is started, just after the computer is booted. A special process, called *init*, is present in the boot image. When it starts running, it reads a file telling how many terminals there are. Then it divides off a new process per terminal. These processes wait for someone to log in. If a login is successful, the login process executes a shell to accept commands. These commands may start up more processes, and so forth. Thus, all the processes in the whole system belong to a single tree, with *init* at the root.

In contrast, Windows has no concept of a process hierarchy. All processes are equal. The only hint of a process hierarchy is that when a process is created, the parent is given a special token (called a **handle**) that it can use to control the child. However, it is free to pass this token to some other process, thus invalidating the hierarchy. Processes in UNIX cannot disinherit their children.

Implementation on Process

To implement the process model, the operating system maintains a table (an array of structures), called the **process table**, with one entry per process. (Some authors call these entries **process control blocks**.) This entry contains important information about the process' state, including its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from *running* to *ready* or *blocked* state so that it can be restarted later as if it had never been stopped.

Figure below shows some of the key fields in a typical system. The fields in the first column relate to process management. The other two relate to memory management and file management, respectively. It should be noted that precisely which fields the process table has is highly system dependent, but this figure gives a general idea of the kinds of information needed.

Now that we have looked at the process table, it is possible to explain a little more about how the illusion of multiple sequential processes is maintained on one (or each) CPU. Associated with each I/O class is a location (typically at a fixed location near the bottom of memory) called the **interrupt vector**. It contains the address of the interrupt service procedure. Suppose that user process 3 is running when a disk interrupt happens. User process 3's program counter, program status word, and sometimes one or more registers are pushed onto the (current) stack by the interrupt hardware. The computer then jumps to the address specified in the interrupt vector. That is all the hardware does. From here on, it is up to the software, in particular, the interrupt service procedure.

All interrupts start by saving the registers, often in the process table entry for the current process. Then the information pushed onto the stack by the interrupt is removed and the stack pointer is set to point to a temporary stack used by the process handler. Actions such as saving the registers and setting the stack pointer cannot even be expressed in high-level languages such as C, so they are performed by a small assembly-language routine, usually the same one for all interrupts since the work of saving the registers is identical, no matter what the cause of the interrupt is.

When this routine is finished, it calls a C procedure to do the rest of the work for this specific interrupt type. (We assume the operating system is written in C, the usual choice for all real operating systems.) When it has done its job, possibly making some process now ready, the scheduler is called to see who to run next. After that, control is passed back to the assembly-language code to load up the registers and memory map for the now-current process and start it running.

Process management	**Memory management**	**File management**
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Fig: Fields of Process Table Entry

Cooperating Processes

Cooperating processes are those that can affect or are affected by other processes running on the system. Cooperating processes may share data with each other.

There may be many reasons for the requirement of cooperating processes. Some of these are given as follows −

- **Modularity**

  Modularity involves dividing complicated tasks into smaller subtasks. These subtasks can completed by different cooperating processes. This leads to faster and more efficient completion of the required tasks.

- **Information Sharing**

  Sharing of information between multiple processes can be accomplished using cooperating processes. This may include access to the same files. A mechanism is required so that the processes can access the files in parallel to each other.

- **Convenience**

  There are many tasks that a user needs to do such as compiling, printing, editing etc. It is convenient if these tasks can be managed by cooperating processes.

- **Computation Speedup**

  Subtasks of a single task can be performed parallely using cooperating processes. This increases the computation speedup as the task can be executed faster. However, this is only possible if the system has multiple processing elements.

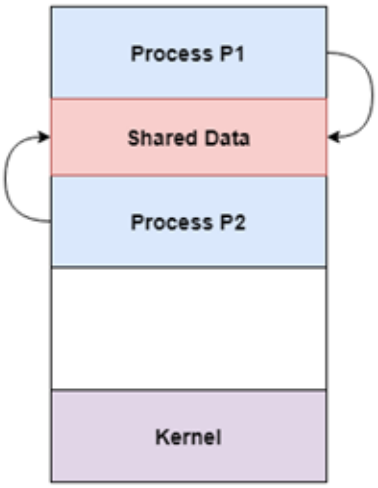
Methods of Cooperation

Cooperating processes can coordinate with each other using shared data or messages. Details about these are given as follows −

- **Cooperation by Sharing**

  The cooperating processes can cooperate with each other using shared data such as memory, variables, files, databases etc. Critical section is used to provide data integrity and writing is mutually exclusive to prevent inconsistent data.

  A diagram that demonstrates cooperation by sharing is given as follows −

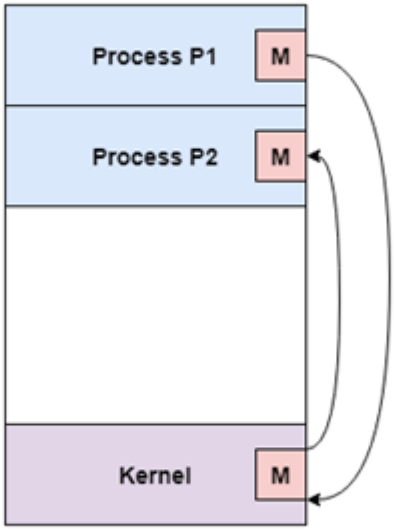


  In the above diagram, Process P1 and P2 can cooperate with each other using shared data such as memory, variables, files, databases etc.

- **Cooperation by Communication**

  The cooperating processes can cooperate with each other using messages. This may lead to deadlock if each process is waiting for a message from the other to perform a operation. Starvation is also possible if a process never receives a message.

  A diagram that demonstrates cooperation by communication is given as follows −



In the above diagram, Process P1 and P2 can cooperate with each other using messages to communicate.

System Calls

The interface between a process and an operating system is provided by system calls. In general, system calls are available as assembly language instructions. They are also included in the manuals used by the assembly level programmers. System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call.

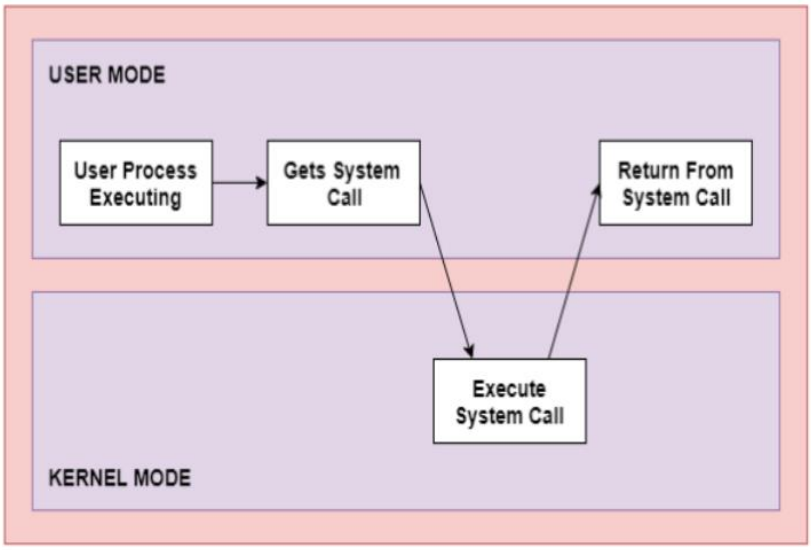


Fig: Execution of System call

As can be seen from this diagram, the processes execute normally in the user mode until a system call interrupts this. Then the system call is executed on a priority basis in the kernel mode. After the execution of the system call, the control returns to the user mode and execution of user processes can be resumed.

In general, system calls are required in the following situations −

- If a file system requires the creation or deletion of files. Reading and writing from files also require a system call.
- Creation and management of new processes.
- Network connections also require system calls. This includes sending and receiving packets.
- Access to a hardware devices such as a printer, scanner etc. requires a system call.

Types of System Calls

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

Process Control

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

File Management

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

Device Management

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

Information Maintenance

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

Communication

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

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

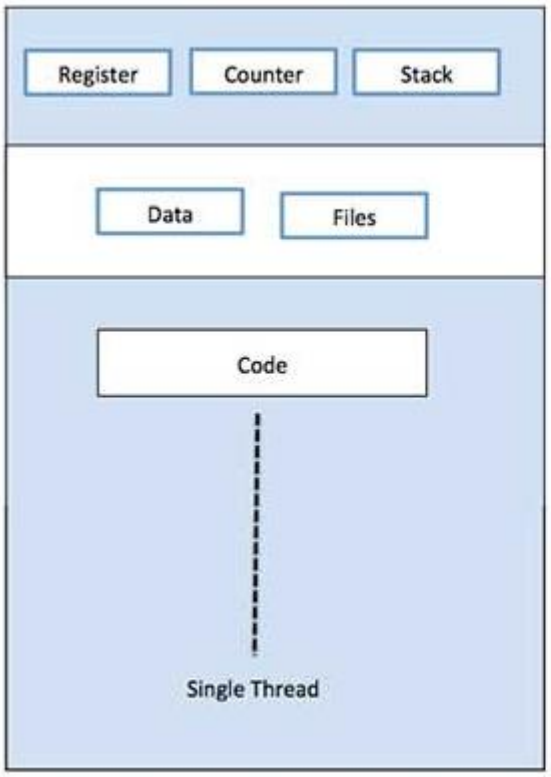
Categories	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
Device manipulation	SetConsoleMode() ReadConsole() WriteConsole()	loctl() read() write()
File manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	Open() Read() write() close!)
Information maintanence	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	Pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup ()	Chmod() Umask() Chown()

Threads

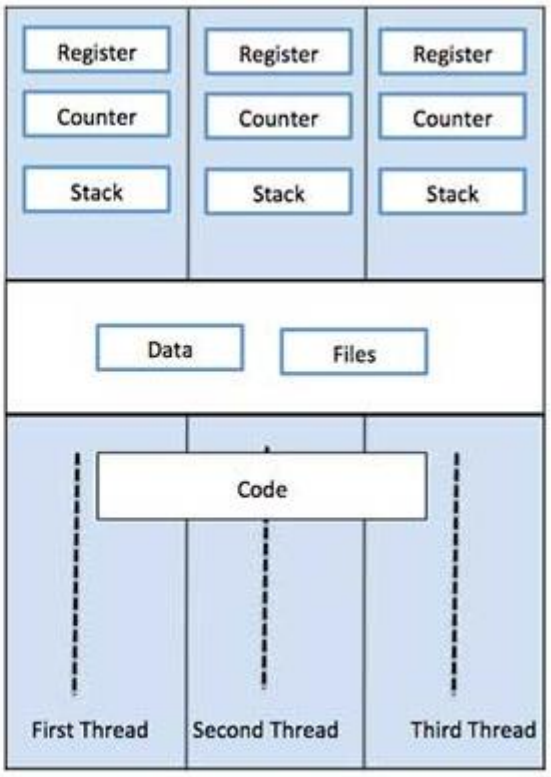
A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism.



Single Process P with single thread



Single Process P with three threads

Difference between process and thread

Process	Thread
1. Process is heavy weight or resource intensive.	1. Thread is light weight, taking lesser resources than a process.
2. Process switching needs interaction with operating system.	2. Thread switching does not need to interact with operating system.
3. In multiple processing environments, each process executes the same code but has its own memory and file resources.	3. All threads can share same set of open files, child processes.
4. If one process is blocked, then no other process can execute until the first process is unblocked.	4. While one thread is blocked and waiting, a second thread in the same task can run

5. Multiple processes without using threads use more resources.	5. Multiple threaded processes use fewer resources.
6. In multiple processes each process operates independently of the others.	6. One thread can read, write or change another thread's data.

Advantages of Thread

1. *Responsiveness:* If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
2. *Faster context switch:* Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.
3. *Effective utilization of multiprocessor system:* If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.
4. *Resource sharing:* Resources like code, data, and files can be shared among all threads within a process.
   Note: stack and registers can't be shared among the threads. Each thread has its own stack and registers.
5. *Communication:* Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two process.

Types of Thread

1. User Level Thread

The user-level threads are implemented by users and the kernel is not aware of the existence of these threads. It handles them as if they were single-threaded processes. User-level threads are small and much faster than kernel level threads. They are represented by a program counter (PC), stack, registers and a small process control block. Also, there is no kernel involvement in synchronization for user-level threads.

Advantages

- User-level threads are easier and faster to create than kernel-level threads. They can also be more easily managed.
- User-level threads can be run on any operating system.
- There are no kernel mode privileges required for thread switching in user-level threads.

Disadvantages

1. Multithreaded applications in user-level threads cannot use multiprocessing to their advantage.
2. The entire process is blocked if one user-level thread performs blocking operation.

2. **Kernel Level Thread**

Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel. The context information for the process as well as the process threads is all managed by the kernel. Because of this, kernel-level threads are slower than user-level threads.

Advantages

1. Multiple threads of the same process can be scheduled on different processors in kernel-level threads.
2. The kernel routines can also be multithreaded.
3. If a kernel-level thread is blocked, another thread of the same process can be scheduled by the kernel.

Disadvantages

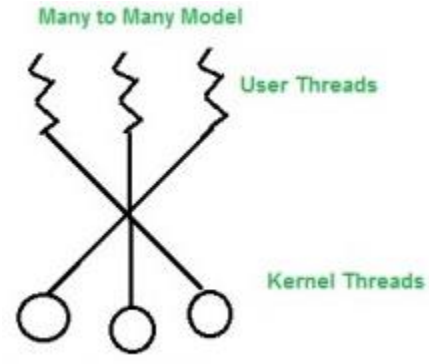
- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

Multithreading Models

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

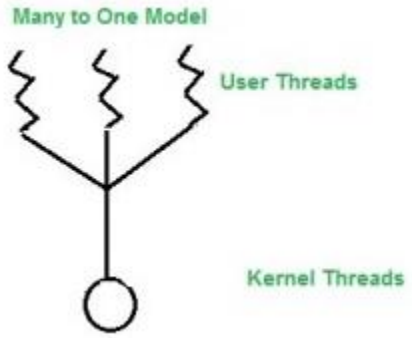
1. **Many to Many relationship.**

Many to many multithreading models in operating system maps many user threads to a lesser or equal number of kernel 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.



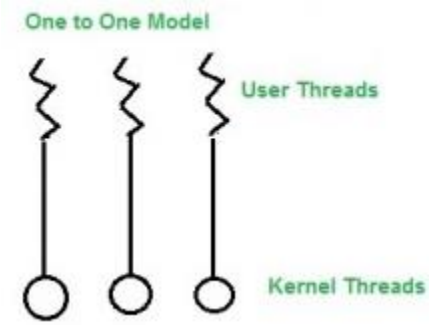
2. **Many to One relationship.**

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.



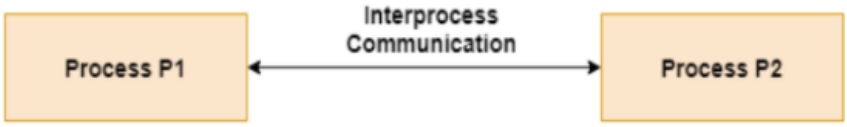
3. **One to One relationship.**

One to one multithreading model maps one user thread to one kernel thread. So, for each user thread, there is a corresponding kernel thread present in the kernel. In this model, thread management is done at the kernel level and the user threads are fully supported by the kernel. Here, if one thread makes the blocking system call, the other threads still run.



Interprocess Communication and Synchronization

Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.



Synchronization is a necessary part of interprocess communication. It is either provided by the interprocess control mechanism or handled by the communicating processes. Some of the methods to provide synchronization are as follows –

- **Semaphore**

  A semaphore is a variable that controls the access to a common resource by multiple processes. The two types of semaphores are binary semaphores and counting semaphores.

- **Mutual Exclusion**

  Mutual exclusion requires that only one process thread can enter the critical section at a time. This is useful for synchronization and also prevents race conditions.

- **Barrier**

  A barrier does not allow individual processes to proceed until all the processes reach it. Many parallel languages and collective routines impose barriers.

- **Spinlock**

  This is a type of lock. The processes trying to acquire this lock wait in a loop while checking if the lock is available or not. This is known as busy waiting because the process is not doing any useful operation even though it is active.

Race Condition

A race condition occurs when two or more threads can access shared data and they try to change it at the same time.

To see how interprocess communication works in practice, let us now consider a simple but common example: a print spooler. When a process wants to print a file, it enters the file name in a special **spooler directory**. Another process, the **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.

Imagine that our spooler directory has a very large number of slots, numbered 0 , 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables, *out*, which points to the next file to be printed, and *in*, which points to the next free slot in the directory. These two variables might well be kept in a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes *A* and *B* decide they want to queue a file for printing. This situation is shown in Fig.

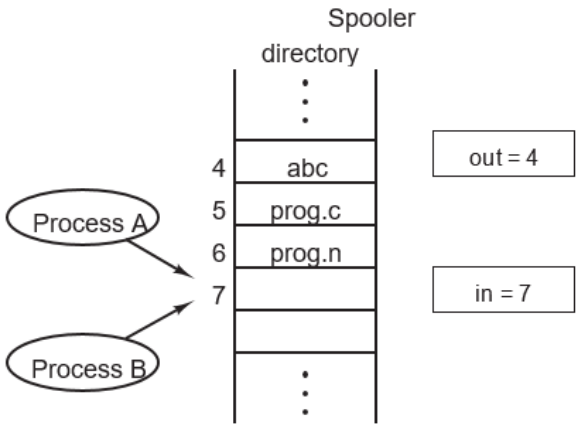


Fig: Two processes want to access shared memory at the same time

Process *A* reads *in* and stores the value, 7, in a local variable called *next free slot*. Just then a clock interrupt occurs and the CPU decides that process *A* has run long enough, so it switches to process *B*. Process *B* also reads *in* and also gets a 7. It, too, stores it in *its* local variable *next free slot*. At this instant both processes think that the next available slot is 7.

Process *B* now continues to run. It stores the name of its file in slot 7 and updates *in* to be an 8. Then it goes off and does other things. Eventually, process *A* runs again, starting from the place it left off. It looks at *next free slot*, finds a 7 there, and writes its file name in slot 7, erasing the name that process *B* just put there. Then it computes *next free slot* + 1, which is 8, and sets *in* to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process *B* will never receive any output. Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race conditions**.

Critical Section

It is part of the program where shared resources are accessed by various processes. When one process is executing in its critical section no other process is to be allowed to execute in its critical section. **Mutual exclusion is** a way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing. If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions.

We need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).

- No two processes may at the same moment inside their critical sections.
- No assumptions are made about relative speeds of processes or number of CPUs.
- No process should outside its critical section should block other processes.
- No process should wait arbitrary long to enter its critical section.

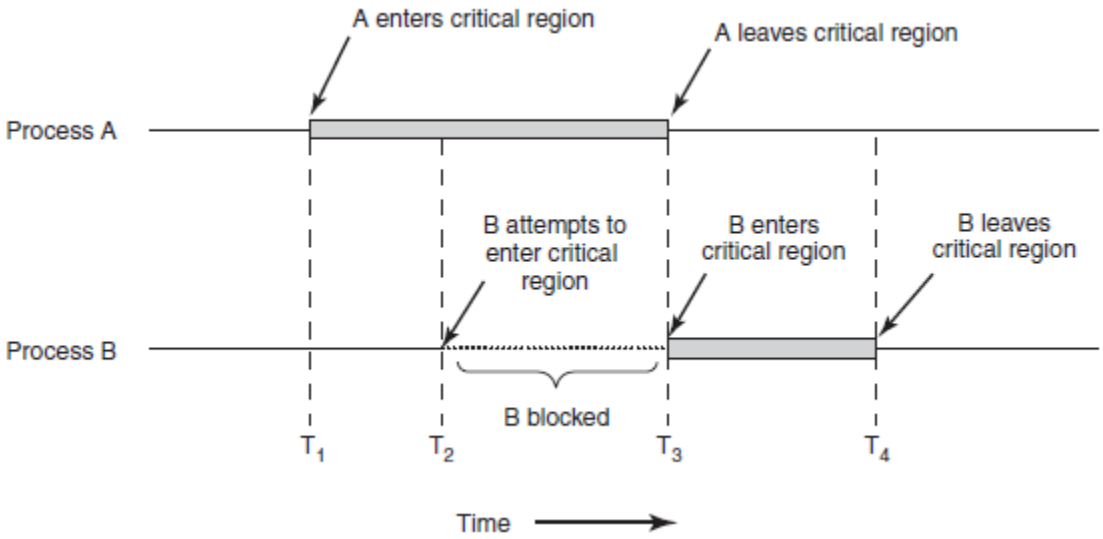


Fig: Mutual exclusion using critical region

Here process *A* enters its critical region at time *T*1. A little later, at time *T*2 process *B* attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, *B* is temporarily suspended until time *T*3 when *A* leaves its critical region, allowing *B* to enter immediately. Eventually *B* leaves (at *T*4) and we are back to the original situation with no processes in their critical regions.

Let us understand the inconsistency in the result while accessing the shared resource simultaneously with the help of an example.

Look at the figure above, suppose there are two processes P0 and P1. Both share a common variable A=0. While accessing A both the processes increments the value of A by 1.

First case

The order of execution of the processes is P0, P1 respectively.

Process P0 reads the value of A=0, increments it by 1 (A=1) and writes the incremented value in A.

Now, process P1 reads the value of A=1, increments its value by 1 (A=2) and writes the incremented value in A.

So, after both the process P0 and P1 finishes accessing the variable A. The value of A is 2.

Second case

Consider that process P0 has read the variable A=0. Suddenly context switch happens and P1 takes the charge, and start executing. P1 would increment the value of A (A=1). After execution P1 gives the charge again to P0.

Now the value of A for P0 is 0 & when it starts executing it would increment the value of A, from 0 to 1.

So here, when both the processes P0 & P1 end up accessing the variable A, the value of A =1 which is different from the value of A=2 in the first case

Now, this type of condition where the sequence of execution of the processes affects the result is called **race condition**

Proposals for achieving mutual exclusion

1. Interrupt disabling

On a single-processor system, the simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

```
DisableInterrupt()
//perform CR task
EnableInterrupt()
```

Advantages

- Mutual exclusion can be achieved by implementing operating system primitives to disable and enable interrupts.

Disadvantages

- It is unwise to give user process the power to turn off interrupts. Suppose that if one of them did it and never turned on again. That could be end of the system.
- If the system is multiprocessor with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction.

2. Lock Variables

When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it

becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Advantages

- Mutual exclusion can be achieved.

Disadvantages

- Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

3. Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

Process 0

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

Process 1

The integer variable *turn*, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects *turn*, finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing *turn* to see when it becomes 1. Continuously testing a variable until some value appears is called **busy waiting**.

When process 0 leaves the critical region, it sets *turn* to 1, to allow process 1 to enter its critical region. Suppose that process 1 finishes its critical region quickly, so that both processes are in their noncritical regions, with *turn* set to 0. Now process 0 executes its whole loop quickly, exiting its critical region and setting *turn* to 1. At this point *turn* is 1 and both processes are executing in their noncritical regions.

Suddenly, process 0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now, because *turn* is 1 and process 1 is busy with its noncritical region. It hangs in its while loop until process 1 sets *turn* to 0.

4. **Peterson Algorithm**

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Before using the shared variables (i.e., before entering its critical region), each process calls *enter region* with its own process number, 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *leave region* to indicate that it is done and to allow the other process to enter, if it so desires.

Initially neither process is in its critical region. Now process 0 calls *enter region*. It indicates its interest by setting its array element and sets *turn* to 0. Since process 1 is not interested, *enter region* returns immediately. If process 1 now makes a call to *enter region*, it will hang there until *interested*[0] goes to *FALSE*, an event that happens only when process 0 calls *leave region* to exit the critical region.

Advantages

- Preserves all conditions of mutual exclusion.

Disadvantages

- Difficulty to program for n process system.

5. **Hardware solution TSL (Test and Set Lock)**

Hardware features can make the programming tasks easier and improve system efficiency.

TSL RX,LOCK

It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock*. The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done. To use the TSL instruction, we will use a shared variable, *lock*, to coordinate access to shared memory.

When *lock* is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory.When it is done, the process sets *lock* back to 0 using an ordinary move instruction.

```
enter_region:
        TSL register, lock      |copy lock to register and set lock to 1
        CMP register, #0        |was lock 0?
        JNE enter_region        |if it was non zero, lock was set, so loop
        RET                     |return to caller;
critical_region();
leave_region:
        MOVE lock, #0           |store a 0 in lock
        RET                     |return to caller
noncritical_section();
```

Advantages

- Preserves all condition, easier programming task and improve system efficiency.

Disadvantages

- Difficulty in hardware design.

6. Sleep and Wakeup

One of the alternative to the busy waiting is to use sleep and wake up pair. Sleep is a system call that causes the caller to block, that is be suspended until another process wakes it up.

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
      int item;

      while (TRUE) {                            /* repeat forever */
            item = produce_item( );             /* generate next item */
            if (count == N) sleep( );           /* if buffer is full, go to sleep */
            insert_item(item);                  /* put item in buffer */
            count = count + 1;                  /* increment count of items in buffer */
            if (count == 1) wakeup(consumer);   /* was buffer empty? */
      }
}


void consumer(void)
{
      int item;

      while (TRUE) {                            /* repeat forever */
            if (count == 0) sleep( );           /* if buffer is empty, got to sleep */
            item = remove_item( );              /* take item out of buffer */
            count = count – 1;                  /* decrement count of items in buffer */
            if (count == N – 1) wakeup(producer); /* was buffer full? */
            consume_item(item);                 /* print item */
      }
}
```

Producer-Consumer problem with a fatal race condition

The producer produces the item and inserts it into the buffer. The value of the global variable count got increased at each insertion. If the buffer is filled completely and no slot is available then the producer will sleep, otherwise it keep inserting. On the consumer's end, the value of count got decreased by 1 at each consumption. If the buffer is empty at any point of time then the consumer will sleep otherwise, it keeps consuming the items and decreasing the value of count by 1. The consumer will be waked up by the producer if there is at least 1 item available in the buffer which is to be consumed. The producer will be waked up by the consumer if there is at least one slot available in the buffer so that the producer can write that.

Types of Mutual Exclusion

Semaphore

Semaphores is integer variable that is used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −

1. **Counting Semaphores**

   These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

The definitions of wait and signal are as follows –

Wait

The wait operation decrements the value of its argument S. If S is negative, then put process in suspend list.

```
wait(S)
{
S--;
if(S<0)
  put process (PCB) in suspend list, sleep()
else
  return;
}
```

Signal

The signal operation increments the value of its argument S.

```
signal(S)
{
S++;
if(S<=0)
  Select a process from  suspend list, wakeup()
}
```

2. **Binary Semaphores**

   The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

The definitions of wait and signal are as follows –

Wait

```
wait(S)
{
if(S==1)
   S=0;
else
  Block this process and place this process in suspend list, sleep()
}
```

Signal

```
signal(S)
{
if(suspend list is empty)
  S=1;
else
  Select a process from suspend list, wakeup()
}
```

Solving Producer Consumer problem using Semaphore

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

To solve this problem, we need two counting semaphores – Full and Empty. “Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.

When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

As the consumer is removing an item from buffer, therefore the value of "full" is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of "empty" by 1. The value of mutex is also increased so that producer can access the buffer now.

Monitors

A monitor is a collection of procedures, variables and data structures grouped together in a single module or package. Processes can call the monitor procedures but cannot access the internal data structures. Only one process can be active in a monitor at any instant. Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter. So, no two processes will be executing their critical regions at the same time.

Monitors are needed because if many programmers are using a semaphore and one programmer forgets to signal after the program has left the critical section, then the entire synchronization mechanism will end up in a deadlock.

```
Monitor monitor_name
   {
     shared variable declarations;
     procedure p1(){ .....  }
     procedure p2(){ ....  }
                    ....
     procedure pn(){ ...}
     {initialization code;}
```

Solving producer-consumer problem using monitors

```
Monitor ProducerConsumer
{
int count;
condition full,empty;
void insert(int item){
        if(count==N) wait(full);
        insert_item(item);
        count++;
        if(count==1)signal(empty);
        }
        void remove(){
        if(count==0) wait(empty);
        remove_item();
        count--;
```

```
if(count==N-1)signal(full);
}
count=0;
};
void producer(){
        while(TRUE){
        item=produce_item();
        ProducerConsumer.insert(item);
        }
   }
 void consumer(){
        while(TRUE){
        item=ProducerConsumer.remove();
        consume_item();
        }
   }
```

Difference between semaphore and monitor

Semaphore	Monitor
1. Semaphores is an integer variable S.	1. Monitor is an abstract data type.
2. The value of Semaphore S indicates the number of shared resources availabe in the system	2. The Monitor type contains shared variables and the set of procedures that operate on the shared variable.
3. When any process access the shared resources it perform wait() operation on S and when it releases the shared resources it performs signal() operation on S.	3. When any process wants to access the shared variables in the monitor, it needs to access it through the procedures.
4. Semaphore does not have condition variables.	4. Monitor has condition variables.

Mutex

Mutex (Mutual Exclusion) lock is essentially a variable that is binary nature that provides code wise functionality for mutual exclusion. At times, there maybe multiple threads that may be trying to access same resource like memory or I/O etc. To make sure that there is no overriding. Mutex provides a locking mechanism.

Only one thread at a time can take the ownership of a mutex and apply the lock. Once it done utilizing the resource and it may release the mutex lock.

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Message Passing

This method of interprocess communication uses two primitives, send and receive, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as

send(destination, &message);

and

receive(source, &message);

The former call sends a message to a given destination and the latter one receives a message from a given source (or from *ANY*, if the receiver does not care). If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

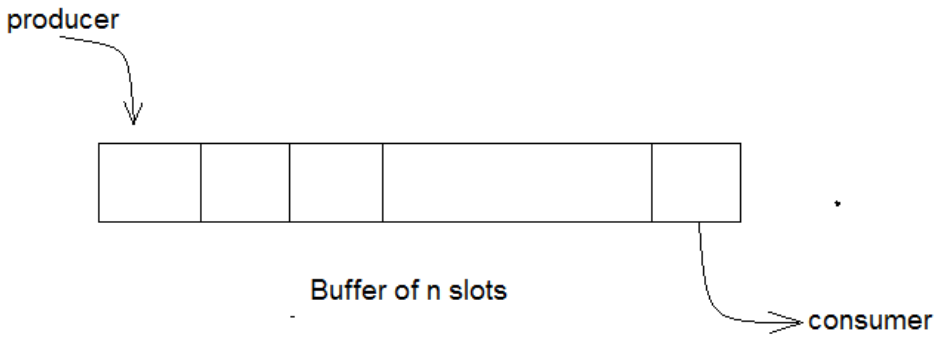
Producer Consumer Problem with Message Passing

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Producer-Consumer Problem (Bounded Buffer)

Bounded Buffer problem which is also called producer consumer problem is one of the classic problems of synchronization.

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

Solution

Producer Operation

a. Producer first waits until there is atleast one empty slot.
b. Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
c. Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
d. After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

Consumer Operation

- The consumer waits until there is at least one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

Serializability

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. Serializability is a concept that helps us to check which schedules are serializable. A serializable schedule is the one that always leaves the database in consistent state.

A serial schedule is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution. However a non-serial schedule needs to be checked for Serializability.

A non-serial schedule of n number of transactions is said to be serializable schedule, if it is equivalent to the serial schedule of those n transactions. A serial schedule doesn't allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

There are two types of serializability

Conflict Serializability is one of the type of Serializability, which can be used to check whether a non-serial schedule is conflict serializable or not. A schedule is called conflict serializable if we can convert it into a serial schedule after swapping its non-conflicting operations.

View Serializability is a process to find out that a given schedule is view serializable or not. To check whether a given schedule is view serializable, we need to check whether the given schedule is View Equivalent to its serial schedule.

```
T1          T2
-----       -----
R(X)
W(X)
            R(X)
            W(X)
R(Y)
W(Y)
            R(Y)
            W(Y)
```

Serial Schedule of the above given schedule:

As we know that in Serial schedule a transaction only starts when the current running transaction is finished. So the serial schedule of the above given schedule would look like this:

```
T1          T2
-----       -----
R(X)
W(X)
R(Y)
W(Y)
            R(X)
            W(X)
            R(Y)
            W(Y)
```

If we can prove that the given schedule is **View Equivalent** to its serial schedule then the given schedule is called **view Serializable**.

Locking Protocols

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

Binary Locks: A Binary lock on a data item can either locked or unlocked states.

Shared/exclusive: This type of locking mechanism separates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

1. Shared Lock (S):

A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.

For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

2. Exclusive Lock (X):

With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.

For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

3. Simplistic Lock Protocol

This type of lock-based protocols allows transactions to obtain a lock on every object before beginning operation. Transactions may unlock the data item after finishing the 'write' operation.

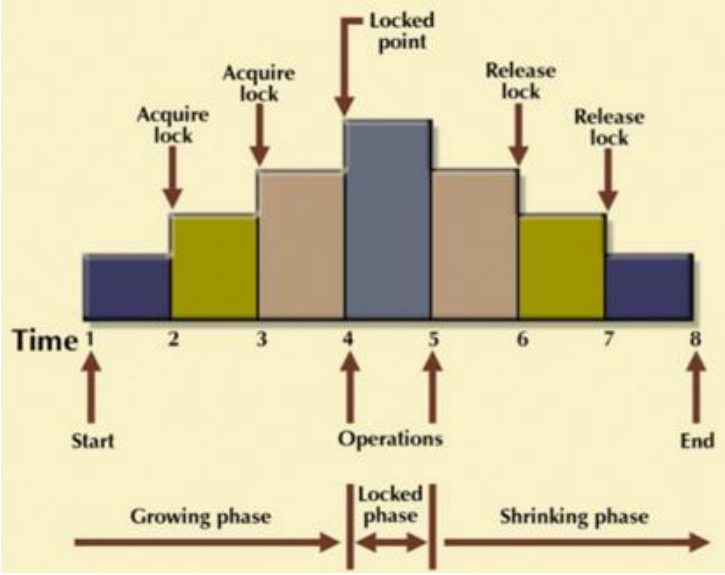
4. Pre-claiming Locking

Pre-claiming lock protocol helps to evaluate operations and create a list of required data items which are needed to initiate an execution process. In the situation when all locks are granted, the transaction executes. After that, all locks release when all of its operations are over.

Two Phase Locking (2PL) Protocol

Two-Phase locking protocol which is also known as a 2PL protocol. It is also called P2L. In this type of locking protocol, the transaction should acquire a lock after it releases one of its locks. This locking protocol divides the execution phase of a transaction into three different parts.

- In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.
- The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.
- In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.



The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:

- **Growing Phase**: In this phase transaction may obtain locks but may not release any locks.
- **Shrinking Phase**: In this phase, a transaction may release locks but not obtain any new lock

It is true that the 2PL protocol offers serializability. However, it does not ensure that deadlocks do not happen.

Timestamp-based Protocols

The timestamp-based algorithm uses a timestamp to serialize the execution of concurrent transactions. This protocol ensures that every conflicting read and write operations are executed in timestamp order. The protocol uses the System Time or Logical Count as a Timestamp.

The older transaction is always given priority in this method. It uses system time to determine the time stamp of the transaction. This is the most commonly used concurrency protocol.

Lock-based protocols help you to manage the order between the conflicting transactions when they will execute. Timestamp-based protocols manage conflicts as soon as an operation is created.

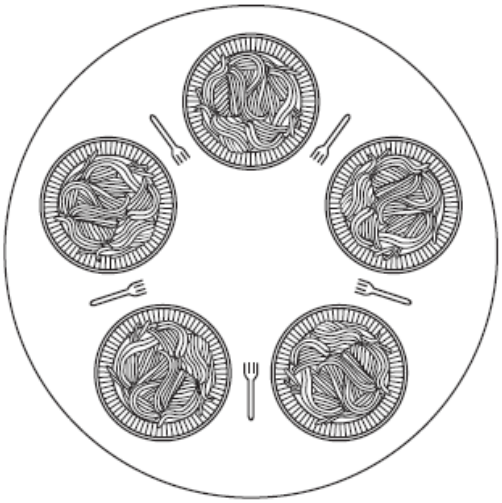
Example:

Suppose there are there transactions T1, T2, and T3.
T1 has entered the system at time 0010
T2 has entered the system at 0020
T3 has entered the system at 0030
Priority will be given to transaction T1, then transaction T2 and lastly Transaction T3.

Classical IPC Problem

Dining philosopher problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the centre of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbours). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbour. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again. The **dining-philosophers problem** is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores.



Although this solution guarantees that no two neighbours are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.
Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

```
#define N           5                       /* number of philosophers */
#define LEFT        (i+N–1)%N               /* number of i's left neighbor */
#define RIGHT       (i+1)%N                 /* number of i's right neighbor */
#define THINKING    0                       /* philosopher is thinking */
#define HUNGRY      1                       /* philosopher is trying to get forks */
#define EATING      2                       /* philosopher is eating */

typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */


void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
      while (TRUE) {                        /* repeat forever */
            think( );                       /* philosopher is thinking */
            take_forks(i);                  /* acquire two forks or block */
            eat( );                         /* yum-yum, spaghetti */
            put_forks(i);                   /* put both forks back on table */
      }
}

void take_forks(int i)                      /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                         /* enter critical region */
      state[i] = HUNGRY;                    /* record fact that philosopher i is hungry */
      test(i);                              /* try to acquire 2 forks */
      up(&mutex);                           /* exit critical region */
      down(&s[i]);                          /* block if forks were not acquired */
}

void put_forks(i)                           /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                         /* enter critical region */
      state[i] = THINKING;                  /* philosopher has finished eating */
      test(LEFT);                           /* see if left neighbor can now eat */
      test(RIGHT);                          /* see if right neighbor can now eat */
      up(&mutex);                           /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

The Readers and Writers Problem

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

Solution
From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one **mutex** m and a **semaphore** w. An integer variable read_count is used to maintain the number of readers currently accessing the resource. The variable read_count is initialized to 0. A value of 1 is given initially to m and w.

Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the read_count variable.

Code for writer process

```
while(TRUE)
{
   wait(w);
  /* perform the write operation */
  signal(w);
}
```

Code for Reader process

```
while(TRUE)
{
   wait(m);              //acquire lock
   read_count++;
   if(read_count == 1)
      wait(w);           //release lock
   signal(m);             /* perform the reading operation */
   wait(m);              // acquire lock
   read_count--;
   if(read_count == 0)
      signal(w);        // release lock
   signal(m);
}
```

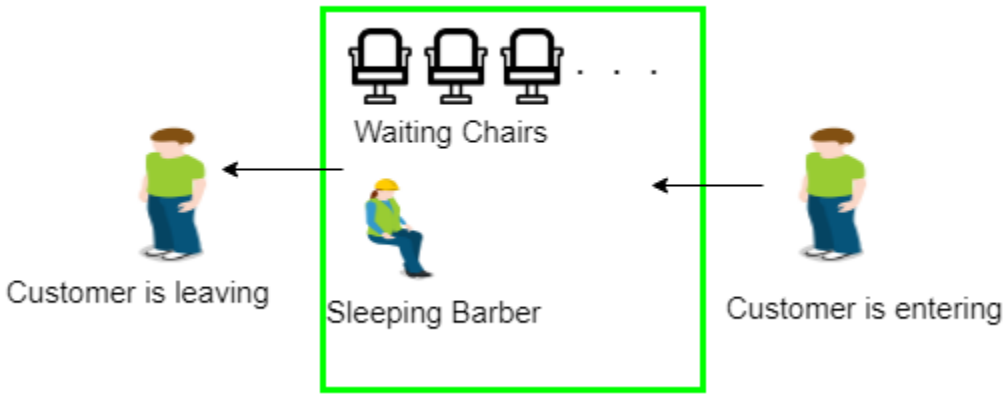
As seen above in the code for the writer, the writer just waits on the w semaphore until it gets a chance to write to the resource. After performing the write operation, it increments w so that the next writer can access the resource. On the other hand, in the code for the reader, the lock is acquired whenever the read_count is updated by a process. When a reader wants to access the resource, first it increments the read_count value, then accesses the resource and then decrements

the read_count value. The semaphore w is used by the first reader which enters the critical section and the last reader which exits the critical section. The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now. Similarly, when the last reader exits the critical section, it signals the writer using the w semaphore because there are zero readers now and a writer can have the chance to access the resource.

The Sleeping Barber Problem

The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.

- If there is no customer, then the barber sleeps in his own chair.
- When a customer arrives, he has to wake up the barber.
- If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.



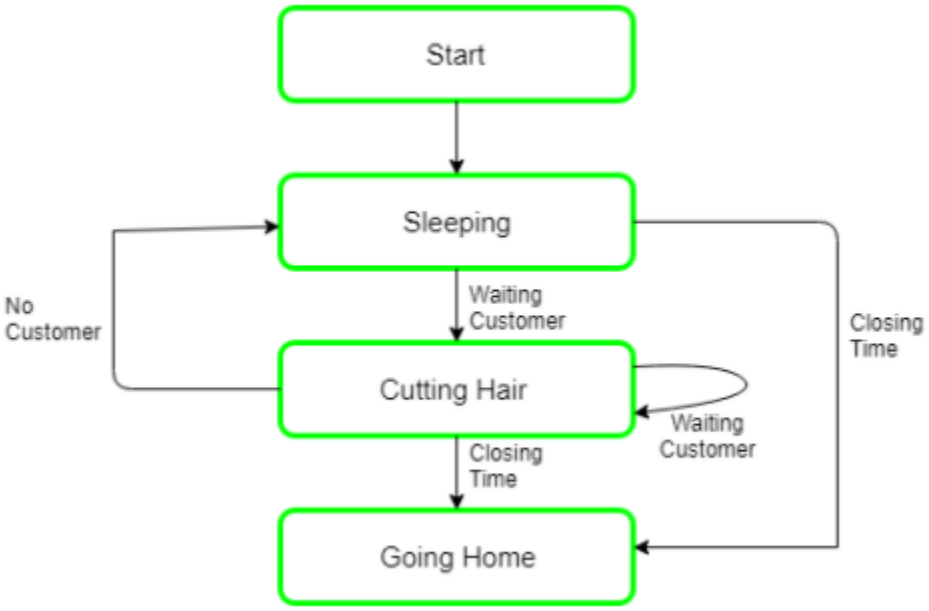
The solution to this problem includes three semaphores. First is for the customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting). Second, the barber 0 or 1 is used to tell whether the barber is idle or is working, And the third mutex is used to provide the mutual exclusion which is required for the process to execute. In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop.

When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. Then the barber goes to sleep until the first customer comes up.

If the chair is available then customer sits in the waiting room and increments the variable waiting value and also increases the customer's semaphore this wakes up the barber if he is sleeping.

At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.

When a customer arrives, he executes customer procedure the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less then the number of chairs then he sits otherwise he leaves and releases the mutex.



Algorithm for Sleeping Barber problems

```
Semaphore Customers = 0;
Semaphore Barber = 0;
Mutex Seats = 1;
int FreeSeats = N;

Barber {
      while(true) {
            /* waits for a customer (sleeps). */
            down(Customers);

            /* mutex to protect the number of available seats.*/
            down(Seats);

            /* a chair gets free.*/
            FreeSeats++;

            /* bring customer for haircut.*/
            up(Barber);

            /* release the mutex on the chair.*/
            up(Seats);
```

```
            /* barber is cutting hair.*/
      }
}

Customer {
      while(true) {
            /* protects seats so only 1 customer tries to sit
               in a chair if that's the case.*/
            down(Seats); //This line should not be here.
            if(FreeSeats > 0) {

                  /* sitting down.*/
                  FreeSeats--;
/* notify the barber. */
                  up(Customers);

                  /* release the lock */
                  up(Seats);

                  /* wait in the waiting room if barber is busy. */
                  down(Barber);
                  // customer is having hair cut
            } else {
                  /* release the lock */
                  up(Seats);
                  // customer leaves
            }
      }
}
```

Process Scheduling

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

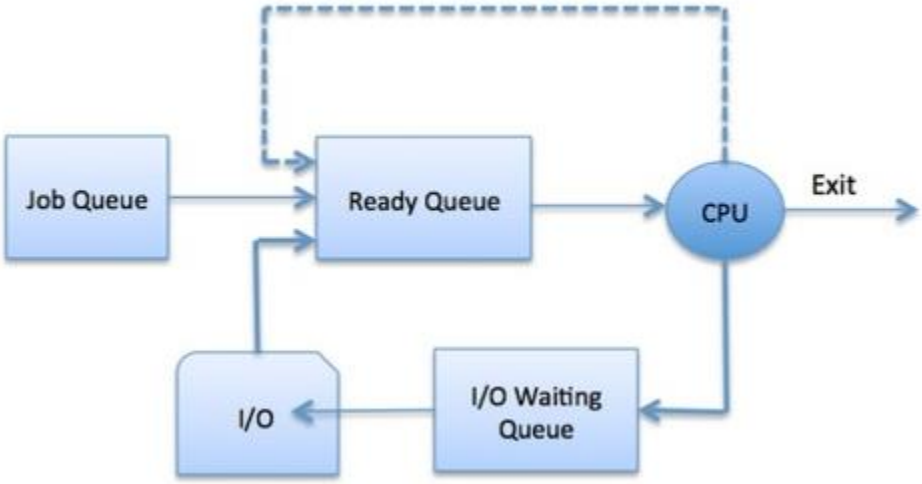
Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues −

- **Job queue** − This queue keeps all the processes in the system.
- **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.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Schedulers

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

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

Long-Term Scheduler

It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long-term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler

Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

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

Dispatcher

A dispatcher is a special program which comes into play after the scheduler. When the scheduler completes its job of selecting a process, it is the dispatcher which takes that process to the desired

state/queue. The dispatcher is the module that gives a process control over the CPU after it has been selected by the short-term scheduler. This function involves the following:

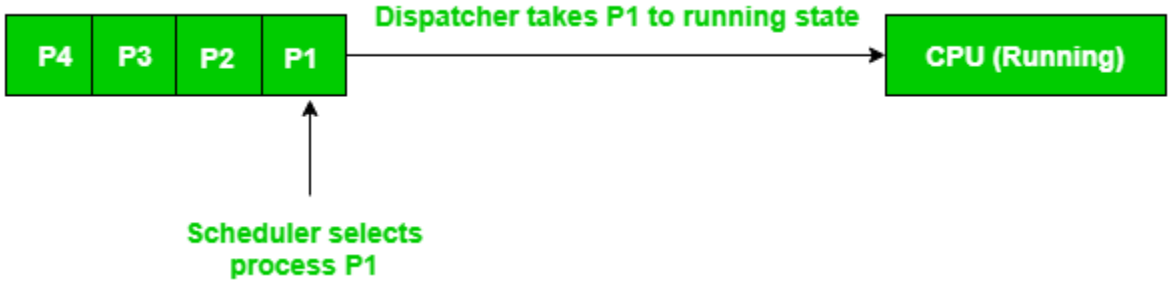
- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

Difference between the Scheduler and Dispatcher

Consider a situation, where various processes are residing in the ready queue waiting to be executed. The CPU cannot execute all of these processes simultaneously, so the operating system has to choose a particular process on the basis of the scheduling algorithm used. So, this procedure of selecting a process among various processes is done by the scheduler. Once the scheduler has selected a process from the queue, the dispatcher comes into the picture, and it is the dispatcher who takes that process from the ready queue and moves it into the running state. Therefore, the scheduler gives the dispatcher an ordered list of processes which the dispatcher moves to the CPU over time.

Example

There are 4 processes in the ready queue, P1, P2, P3, P4; Their arrival times are t0, t1, t2, t3 respectively. A First in First out (FIFO) scheduling algorithm is used. Because P1 arrived first, the scheduler will decide it is the first process that should be executed, and the dispatcher will remove P1 from the ready queue and give it to the CPU. The scheduler will then determine P2 to be the next process that should be executed, so when the dispatcher returns to the queue for a new process, it will take P2 and give it to the CPU. This continues in the same way for P3, and then P4.



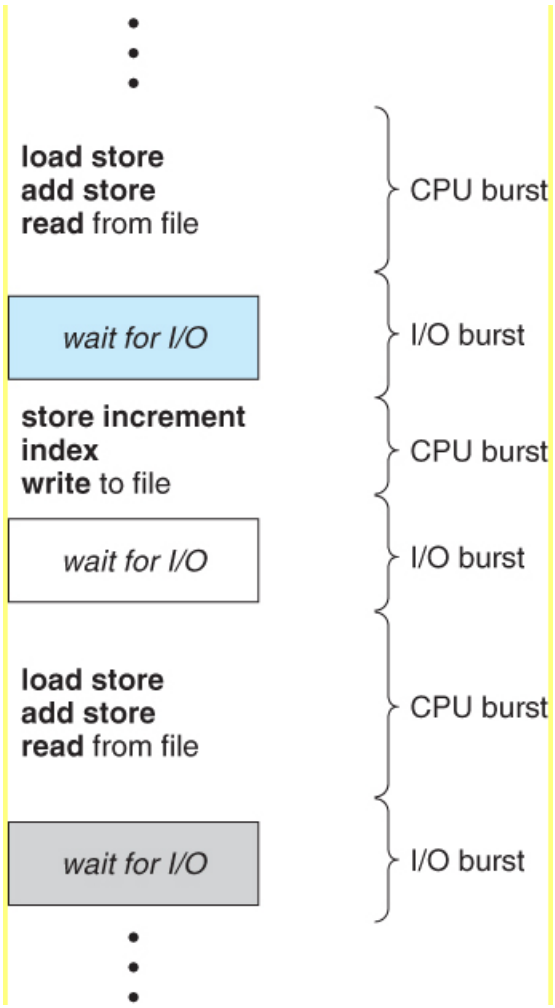
Difference between Dispatcher and Scheduler

Dispatcher	Scheduler
1. Dispatcher is a module that gives control of CPU to the process selected by short term scheduler.	1. Scheduler is something which selects a process among various processes.
2. There are no different types in dispatcher.It is just a code segment.	2. There are 3 types of scheduler i.e. Long-term, Short-term, Medium-term.
3. Working of dispatcher is dependent on scheduler.Means dispatcher have to wait until scheduler selects a process.	3. Scheduler works independently. It works immediately when needed.

4. Dispatcher has no specific algorithm for its implementation.	4. Scheduler works on various algorithm such as FCFS, SJF, RR etc.
5. The time taken by dispatcher is called dispatch latency.	5. Time taken by scheduler is usually negligible. Hence we neglect it.
6. Dispatcher is also responsible for: Context Switching, Switch to user mode, Jumping to proper location when process again restarted.	6. The only work of scheduler is selection of processes.

CPU-I/O Burst Cycle

The success of CPU scheduling depends on the following observed property of processes. Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate back and forth between these two states. The execution begins with CPU burst, followed by I/O burst, then another CPU burst and so on. The last CPU burst will end with a system request to terminate execution rather than with another I/O burst. An I/O bound program would typically have many short CPU bursts; a CPU bound program might have a few very long CPU bursts. The duration of these CPU bursts are measured, which help to select an appropriate CPU scheduling algorithm.



Context Switch

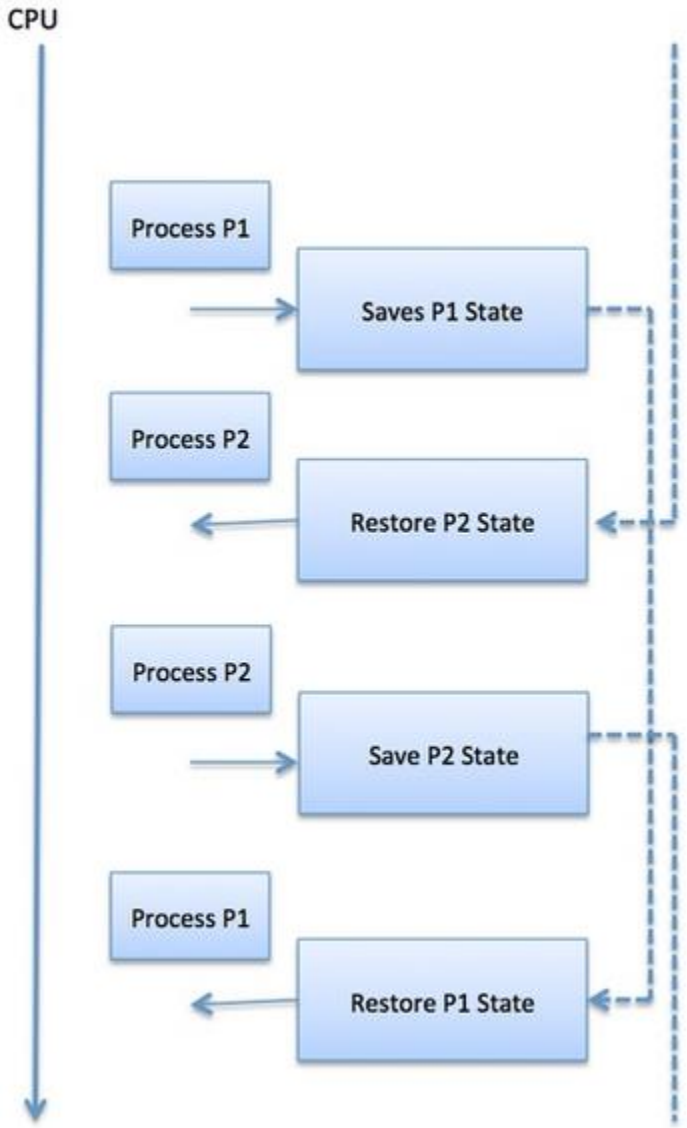
A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

- Program Counter

- Scheduling information
- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information



Types of Scheduling

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

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

In circumstances 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however in circumstances 2 and 3.

When Scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is **non-preemptive**; otherwise the scheduling scheme is **preemptive**.

Preemptive Scheduling

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

Preemptive scheduling is used when a process switches from running state to ready state or from waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for the limited amount of time and then is taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in ready queue till it gets next chance to execute.

Advantages

- Preemptive scheduling method is more robust, approach so one process cannot monopolize the CPU
- Choice of running task reconsidered after each interruption.
- Each event cause interruption of running tasks
- The OS makes sure that CPU usage is the same by all running process.
- In this, the usage of CPU is the same, i.e., all the running processes will make use of CPU equally.
- This scheduling method also improvises the average response time.
- Preemptive Scheduling is beneficial when we use it for the multi-programming environment.

Disadvantages

- Need limited computational resources for Scheduling
- Takes a higher time by the scheduler to suspend the running task, switch the context, and dispatch the new incoming task.
- The process which has low priority needs to wait for a longer time if some high priority processes arrive continuously.

Non Preemptive Scheduling

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems.

It is the only method that can be used on certain hardware platforms, because It does not require the special hardware (for example: a timer) needed for preemptive scheduling.

Non-preemptive Scheduling is used when a process terminates, or a process switches from running to waiting state. In this scheduling, once the resources (CPU cycles) is allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state. In case of non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits till the process complete its CPU burst time and then it can allocate the CPU to another process.

Difference between Preemptive and Non Preemptive Scheduling

Preempted	Non Preemptive
1. A processor can be preempted to execute the different processes in the middle of any current process execution.	1. Once the processor starts its execution, it must finish it before executing the other. It can't be paused in the middle.
2. CPU utilization is more efficient compared to Non-Preemptive Scheduling.	2. CPU utilization is less efficient compared to preemptive Scheduling.
3. Waiting and response time of preemptive Scheduling is less.	3. Waiting and response time of the non-preemptive Scheduling method is higher.
4. Preemptive Scheduling is prioritized. The highest priority process is a process that is currently utilized.	4. When any process enters the state of running, the state of that process is never deleted from the scheduler until it finishes its job.
5. Preemptive Scheduling is flexible.	5. Non-preemptive Scheduling is rigid.
6. Examples: - Shortest Remaining Time First, Round Robin, etc.	6. Examples: First Come First Serve, Shortest Job First, Priority Scheduling, etc.
7. Preemptive Scheduling algorithm can be pre-empted that is the process can be Scheduled	7. In non-preemptive scheduling process cannot be Scheduled

8. In this process, the CPU is allocated to the processes for a specific time period.	8. In this process, CPU is allocated to the process until it terminates or switches to the waiting state.
9. Preemptive algorithm has the overhead of switching the process from the ready state to the running state and vice-versa.	9. Non-preemptive Scheduling has no such overhead of switching the process from running into the ready state.

Scheduling Criteria

There are many different criteria to check when considering the **"best"** scheduling algorithm, they are:

CPU Utilization

To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

Throughput

It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

Turnaround Time

It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process (Wall clock time).

Waiting Time

The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

Load Average

It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

Response Time

Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution (final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

Scheduling Algorithm

Scheduling algorithms are mainly divided into following three categories.

1. Batch system scheduling.
    a. First Come First Served
    b. Shortest Job First
    c. Shortest Remaining Time Next
2. Interactive system scheduling
    a. Round Robin Scheduling
    b. Priority Scheduling
    c. Multilevel Queue Scheduling
    d. Multilevel Feedback Queue Scheduling
3. Real time scheduling
    a. Priority Fair Share Scheduling
    b. Guaranteed Scheduling
    c. Lottery Scheduling
    d. High Response Ratio Next

1. Batch system Scheduling

Batch processing is a technique in which an Operating System collects the programs and data together in a batch before processing starts. An operating system does the following activities related to batch processing.

- The OS defines a job which has predefined sequence of commands, programs and data as a single unit.
- The OS keeps a number a job in memory and executes them without any manual information.
- Jobs are processed in the order of submission, i.e first come first served fashion.
- When a job completes its execution, its memory is released and the output for the job gets copied into an output spool for later printing or processing.

a. First Come First Served

FCFS provides an efficient, simple and error-free process scheduling algorithm that saves valuable CPU resources. It uses nonpreemptive scheduling in which a process is automatically queued and processing occurs according to an incoming request or process order. FCFS derives its concept from real-life customer service.

Let's take a look at how FCFS process scheduling works. Suppose there are three processes in a queue: P1, P2 and P3. P1 is placed in the processing register with a waiting time of zero seconds and 10 seconds for complete processing. The next process, P2, must wait 10 seconds and is placed in the processing cycle until P1 is processed. Assuming that P2 will take 15 seconds to complete, the final process, P3, must wait 25 seconds to be processed. FCFS may not be the fastest process scheduling algorithm, as it does not check for priorities associated with processes. These priorities may depend on the processes' individual execution times.

Characteristics

- Processes are scheduled in the order they are received.

- Once the process has the CPU, it runs to completion.
- Easily implemented, by managing a simple queue or by storing time the process was received.
- Fair to all processes.

Problems

- No guarantee of good response time.
- Large average waiting time.

Example 1

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using the FCFS scheduling algorithm.

Process	Burst time
P1	21
P2	3
P3	6
P4	2

Solution

Gannt chart for the above processes

P1	**P2**	**P3**	**P4**

0 21 24 30 32

Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	21-0 =21 ms	21-21 = 0
P2	24-0 =24 ms	24-3 = 21
P3	30-0 = 30 ms	30-6 = 24
P4	32-0 = 32 ms	32-2 = 30

Total waiting time: (0+21+24+30) = 75 ms

Average waiting time: 75/4 = 18.75 ms

Total turnaround time: (21+24+30+32) = 107 ms

Average turnaround time: 107/4 = 26.75 ms

Example 2

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, 2, 2 and 3 respectively and given Burst Time, let's find the average waiting time using the FCFS scheduling algorithm.

Process	Arrival time	Burst time
P1	0 ms	21
P2	2 ms	3
P3	2 ms	6
P4	3 ms	2

Solution

Gannt chart for the above processes

P1	**P2**	**P3**	**P4**

0 21 24 30 32

Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	21-0 =21 ms	21-21 = 0
P2	24-2 =22 ms	22-3 = 19
P3	30-2 = 28 ms	28-6 = 22
P4	32-3 = 29 ms	29-2 = 27

Total waiting time: (0+19+22+27) = 68 ms

Average waiting time: 68/4 = 17 ms

Total turnaround time: (21+22+28+29) = 100 ms

Average turnaround time: 100/4 = 25 ms

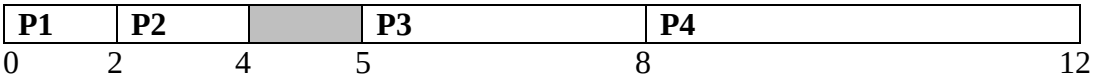
Example 3

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order with Arrival Time 0, 1, 5 and 6 respectively and given Burst Time. let's find the average waiting time using the FCFS scheduling algorithm.

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

Solution

Gannt chart for the above processes



Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	2-0 =2 ms	2-2 = 0
P2	4-1 =3 ms	3-2 = 1
P3	8-5 = 3 ms	3-3 = 0
P4	12-6 = 6 ms	6-4 = 2

Total waiting time: (0+1+0+2) = 3 ms

Average waiting time: 3/4 = 0.75 ms

Total turnaround time: (2+3+3+6) = 14 ms

Average turnaround time: 14/4 = 3.5 ms

b. Shortest job first

In the FCFS, we saw if a process is having a very high burst time and it comes first then the other process with a very low burst time have to wait for its turn. So, to remove this problem, we come with a new approach i.e. Shortest Job First or SJF.
In this technique, the process having the minimum burst time at a particular instant of time will be executed first. It is a non-preemptive approach i.e. if the process starts its execution then it will be fully executed and then some other process will come.
Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next.

Characteristics

- The processing time are known in advanced.
- SJF selects the process with shortest expected processing time. In case of the tie FCFS scheduling is used.
- The decision policies are based on the CPU burst time.

Advantages

- Reduces the average waiting time over FCFS.
- Favors shorts jobs at the cost of long jobs.

Problems

- It may lead to starvation if only short burst time processes are coming in the ready state.
- Estimation of run time to completion.
- Accuracy
- Not applicable in timesharing system.

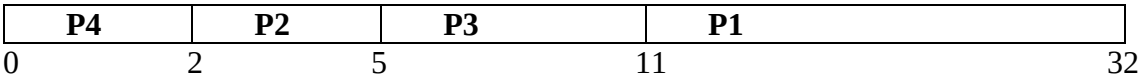
Example 1

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using the SJF scheduling algorithm.

Process	Burst time
P1	21
P2	3
P3	6
P4	2

Solution

Gantt chart for the above processes



As in the Gantt chart, the process P4 will be picked up first as it has the shortest burst time, then P2, followed by P3 and at last P1.

Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	32-0 = 32 ms	32-21 = 11
P2	5-0 =5 ms	5-3 = 2
P3	11-0 = 11 ms	11-6 = 5
P4	2-0 =2 ms	2-2 = 0

Total waiting time: (0+2+5+11) = 18 ms

Average waiting time: 18/4 = 4.5 ms

Total turnaround time: (2+5+11+32) = 50 ms

Average turnaround time: 50/4 = 12.5 ms

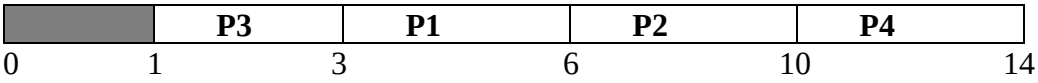
Example 2

Consider the processes P1, P2, P3, P4 given in the below table with arrival time 1, 2, 1 and 4 respectively and given Burst Time, let's find the average waiting time using the SJF scheduling algorithm.

Process	Arrival Time	Burst time
P1	1	3
P2	2	4
P3	1	2
P4	4	4

Solution

Gantt chart for the above processes



Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P3	3-1 =2 ms	2 -2 = 0
P1	6-1 =5 ms	5-3 = 2
P2	10-2 = 8 ms	8-4 = 4
P4	14-4 = 10 ms	10-4 = 6

Total waiting time: (0+2+4+6) = 12 ms

Average waiting time: 12/4 = 3 ms

Total turnaround time: (2+5+8+10) = 25 ms

Average turnaround time: 25/4 = 6.25 ms

c. Shortest Remaining Time Next

It is a preemptive version of shortest job next scheduling. In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Shortest remaining time is advantageous because short processes are handled very quickly. The system also requires very little overhead since it only makes a decision when a process completes or a new process is added, and a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.

Characteristics

- Low average waiting time than shortest job first.
- Useful in timesharing

Problem

- Very high overhead than shortest job first.
- Requires additional computation.
- Favors short jobs, long jobs can be victims of starvation.

Example 1

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, 1, 2 and 3 respectively and given Burst Time, let's find the average waiting time and average turnaround time using the shortest remaining time next scheduling algorithm.

Process	Arrival time	Burst time
P1	0 ms	21
P2	1 ms	3
P3	2 ms	6
P4	3 ms	2

Solution

Gantt chart for the above processes

P1	**P2**	**P4**	**P2**	**P3**	**P1**

0 1 3 5 6 12 32

As it is seen in the Gantt chart above, as P1 arrives first, hence it's execution starts immediately, but just after 1 ms, process P2 arrives with a burst time of 3 ms which is less than the burst time of P1, hence the process P1 (1 ms done, 20 ms left) is preempted and process P2 is executed.

As P2 is getting executed, after 1ms, P3 arrives, but it has a burst time greater than that of P2, hence execution of P2 continues. But after another millisecond, P4 arrives with a burst time of 2ms, as a result P2 (2ms done, 1ms left) is preempted and P4 is executed.

After the completion of P4, process P2 is picked up and finishes, then P3 will get executed and at last P1.

Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	32-0 =32 ms	32 -21 = 11
P2	6-1 =5 ms	5-3 = 2
P3	12-2 = 10 ms	10-6 = 4
P4	5-3 = 2 ms	2-2 = 0

Total waiting time: (11+2+4+0) = 17 ms

Average waiting time: 17/4 = 4.25 ms

Total turnaround time: (32+5+10+2) = 49 ms

Average turnaround time: 49/4 = 12.25 ms

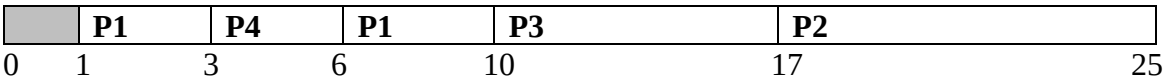
Example 2

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 1, 1, 2 and 3 respectively and given Burst Time, let's find the average waiting time and average turnaround time using the shortest remaining time next scheduling algorithm.

Process	Arrival time	Burst time
P1	1 ms	6
P2	1 ms	8
P3	2 ms	7
P4	3 ms	3

Solution

Gantt chart for the above processes



In the above example, at time 1ms, there are two processes i.e. P1 and P2. Process P1 is having burst time as 6ms and the process P2 is having 8ms. So, P1 will be executed first. Since it is a preemptive approach, so we have to check at every time quantum. At 2ms, we have three processes i.e. P1 (5ms remaining), P2 (8ms) and P3 (7ms). Out of these three, P1 is having the least burst time, so it will continue its execution. After 3ms, we have four processes i.e. P1 (4ms remaining), P2 (8ms), P3 (7ms) and P4 (3ms). Out of these four, P4 is having the least burst time, so it will be executed. The process P4 keeps on executing for the next 3ms because it is having the shortest burst time. After 6ms we have 3 processes i.e P1 (4ms remaining), P2 (8ms), and P3 (7ms). So P1 will be selected and executed. This process of time comparison will continue until we have all the processes executed. So waiting and turnaround time of the processes will be:

Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	10-1 =9 ms	9 -6 = 3
P2	25-1 =24 ms	24-8 = 16
P3	17-2 = 15 ms	15-7 = 8
P4	6-3 = 3 ms	3-3 = 0

Total waiting time: (3+16+8+0) = 27 ms

Average waiting time: 27/4 = 6.75 ms

Total turnaround time: (9+24+15+3) = 51 ms

Average turnaround time: 51/4 = 12.75 ms

2. Interactive System Scheduling

a. Round Robin Scheduling

In this algorithm the process is allocated the CPU for the specific time period called time slice or quantum, which is normally of 10 to 100 milliseconds. If the process completes its execution within this time slice, then it is removed from the queue otherwise it has to wait for another time slice. Preempted process is placed at the back of the ready list.

In this approach of CPU scheduling, we have a fixed time quantum and the CPU will be allocated to a process for that amount of time only at a time. For example, if we are having three process P1, P2 and P3 and our time quantum is 2ms, then P1 will be given 2ms for its execution, then P2 will be given 2ms, then P3 will be given 2ms. After one cycle, again P1 will be given 2ms, and then P2 will be given 2ms and so on until the processes complete its execution.

Advantages

- Fair allocation of CPU across the process.
- Used in timesharing system.
- Low average waiting time when process lengths vary widely.
- Poor average waiting time when process lengths are identical.

Disadvantages

- We have to perform a lot of context switching.
- Time consuming scheduling for small quantum.

Quantum size: If the quantum is very large, each process is given as much time as needs for completion, Round Robin degenerate to First Come First Serve policy. If quantum is very small, system bus at just switching from one process to another process, the overhead of context-switching causes the system efficiency degrading.

Example 1

Consider the processes P1, P2, P3 given in the below table, arrives for execution in the same order, with Arrival Time 0 and given Burst Time, let's find the average waiting time and average turnaround time using the Round Robin scheduling algorithm with quantum size = 2.

Process	Burst time
P1	10
P2	5
P3	8

Solution

Gantt chart for the above processes

Ready queue	**P1**	**P2**	**P3**	**P1**	**P2**	**P3**	**P1**	**P2**	**P3**	**P1**	**P3**	**P1**

Running queue	**P1**	**P2**	**P3**	**P1**	**P2**	**P3**	**P1**	**P2**	**P3**	**P1**	**P3**	**P1**

0 2 4 6 8 10 12 14 15 17 19 21 23

Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	23-0 =23 ms	23-10 = 13
P2	15-0 =15 ms	15-5 = 10
P3	21-0 = 21 ms	21-8 = 13

Total waiting time: (13+10+13) = 36 ms

Average waiting time: 36/3 = 12 ms

Total turnaround time: (23+15+21) = 59 ms

Average turnaround time: 59/3 = 19.66 ms

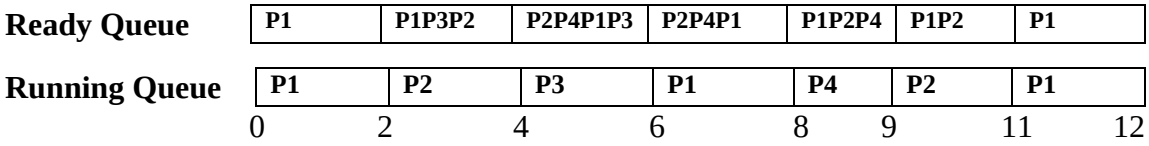
Example 2

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, 1, 2, 4 respectively and given Burst Time, let's find the average waiting time and average turnaround time using the Round Robin scheduling algorithm with quantum size = 2. How many time context switches occurs?

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

Solution

Gantt chart for the above processes



Here context switches occur for 6 times.

Process	Turnaround time = Completion Time - Arrival Time	Waiting time = Turn Around Time – Burst Time	Response Time= CPU First time – Arrival Time
P1	12-0 =12 ms	12-5 = 7	0-0=0
P2	11-1 =10 ms	10-4 = 6	2-1=1
P3	6-2 = 4 ms	4-2 = 2	4-2=2
P4	9-4 = 5 ms	5-1 = 4	8-4=4

Total waiting time: (7+6+2+4) = 19 ms

Average waiting time: 19/4 = 4.75 ms

Total turnaround time: (12+10+4+5) = 31 ms

Average turnaround time: 31/4 = 7.75 ms

Total response time: (0+1+2+4) = 7 ms

Average response time: 7/4 = 1.75 ms

b. Priority scheduling

In this scheduling algorithm the priority is assigned to all the processes and the process with highest priority executed first. Priority assignment of process is done on the basis of internal factor such as CPU and memory requirements or external factor such as user's choice. It is just used to identify which process is having a higher priority and which process is having a lower priority. For example, we can denote 0 as the highest priority process and 100 as the lowest priority process. Also, the reverse can be true i.e we can denote 100 as the highest priority and 0 as the lowest priority.

Advantages

- Highest priority processes like system processes are executed first.

Disadvantages

- It can lead to starvation if only higher priority process comes into the ready state. Low priority processes may never execute.
- If the priorities of more two processes are the same, then we have to use some other scheduling algorithm.

Example 1

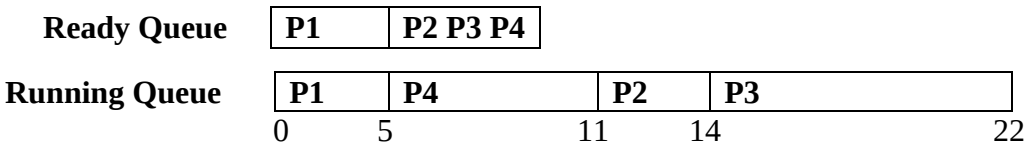
Find average waiting time and average turnaround time of following four processes with their arrival time, burst time and priority as below.

Process	Arrival time	Burst time	Priority
P1	0	5	1
P2	1	3	2
P3	2	8	1
P4	3	6	3

Take higher priority number as high priority.

Solution

Gantt chart for the above processes



In the above example, at 0ms, we have only one process P1. So P1 will execute for 5ms because we are using non-preemption technique here. After 5ms, there are three processes in the ready state i.e. process P2, process P3 and process P4. Out of these three processes, the process P4 is having the highest priority so it will be executed for 6ms and after that, process P2 will be executed for 3ms followed by the process P1. The waiting and turnaround time of processes will be:

Process	Turnaround time = Completion Time - Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	5-0 =5 ms	5-5 = 0
P2	14-1 =13 ms	13-3 = 10
P3	22-2 = 20 ms	20-8 = 12
P4	11-3 = 8 ms	8-6 = 2

Total waiting time: (0+10+12+2) = 24 ms

Average waiting time: 24/4 = 6 ms

Total turnaround time: (5+13+20+8) = 46 ms

Average turnaround time: 46/4 = 11.5 ms

c. Multiple queues (Multilevel Queue Scheduling)

It may happen that processes in the ready queue can be divided into different classes where each class has its own scheduling needs. For example, a common division is a foreground (interactive) process and background (batch) processes. These two classes have different scheduling needs. For this kind of situation Multilevel Queue Scheduling is used. Now let us see how it works.

Ready Queue is divided into separate queues for each class of processes. For example, let us take three different types of process System processes, Interactive processes and Batch Processes. All three processes have their own queue. Now, look at the below figure.

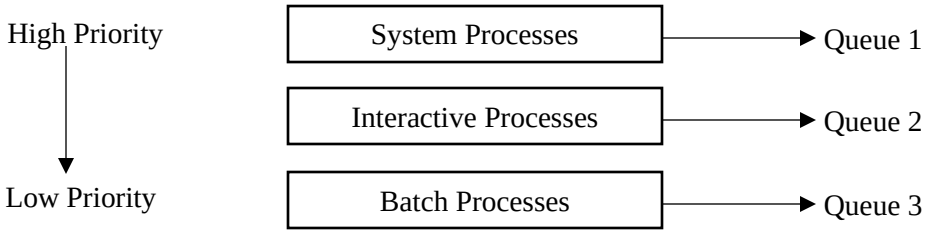


Fig: Multiple queue

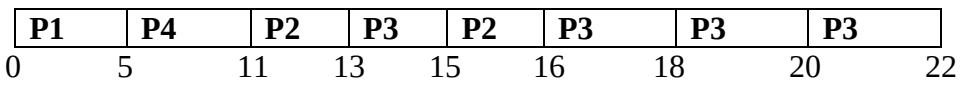
All three different types of processes have their own queue. Each queue has its own Scheduling algorithm. For example, Queue 1 and Queue 2 uses Round Robin while Queue 3 can use FCFS to schedule their processes.

Example : Consider below table of four processes under multilevel queue scheduling. Queue number denotes the queue of the process.

Process	Arrival Time	CPU Brust Time	Queue Number
P1	0	5	1
P2	0	3	2
P3	0	8	2
P4	0	6	1

Solution:

In the above example, we have two queues, i.e Queue 1 and Queue 2. Queue 1 is having higher priority and Queue 1 is using the FCFS approach and Queue 2 is using the round-robin approach (time quantum =2ms). Below is the Gantt chart of the problem.



Since the priority of Queue 1 is higher, so Queue 1 will be executed first. In the Queue 1, we have two processes i.e. P1 and P4 and we are using FCFS. So, P1 will be executed followed by P4.

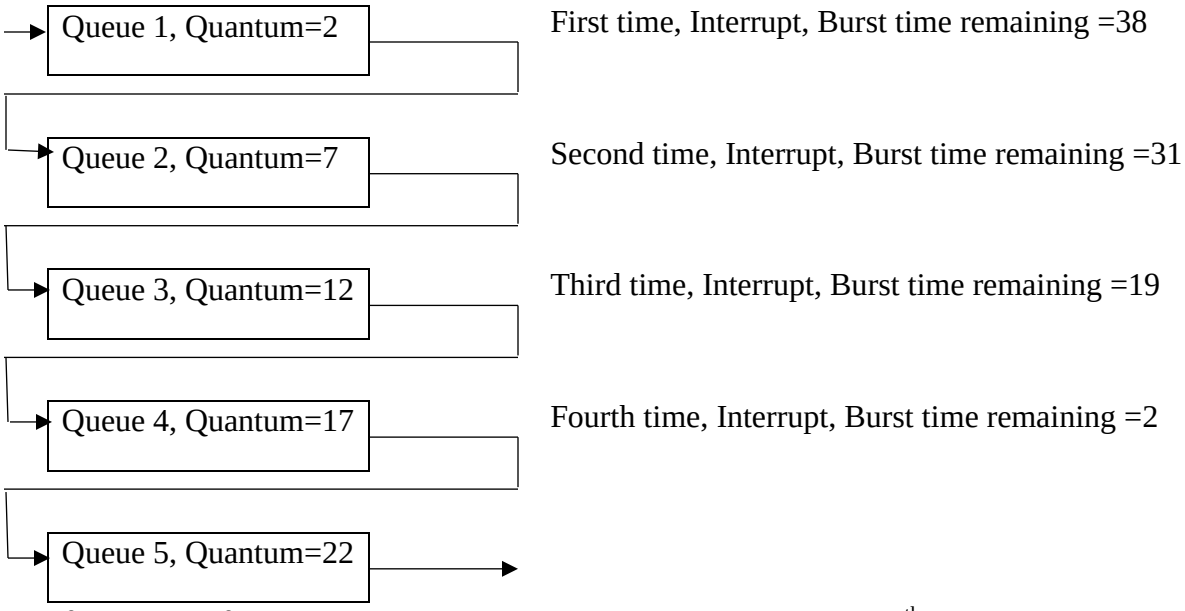
Now, the job of the Queue 1 is finished. After this, the execution of the processes of Queue2 will be started by using the round-robin approach.

d. Multilevel Feedback Queue Scheduling

It allows the process to move in between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it is moved to a lower-priority queue. Similarly, a process that waits too long in a lower-priority queue may be moved to a higher priority queue. This form of aging prevents starvation.

Example: Consider a system which has CPU Bound process which requires burst time of 40 times units. Multilevel feedback queue scheduling is used. The time quantum is 2 units and it will be incremented by 5 units in each level. How many times the process will be interrupted and in which queue process will complete the execution?

Solution:



Hence from above figure process will be interrupted in 4 times and in 5^{th} queue the process will complete the execution.

3. Realtime System Scheduling

A real-time scheduling system is composed of the scheduler, clock and the processing hardware elements. In a real-time system, a process or task has schedulability; tasks are accepted by a real-time system and completed as specified by the task deadline depending on the characteristics of the scheduling algorithm. Modeling and evaluation of a real-time scheduling system concern is on the analysis of the algorithm capability to meet a process deadline. A deadline is defined as the time required for a task to be processed. For example, in a real-time scheduling algorithm a deadline could be set to 5 nano-seconds. In a critical operation the task must be processed in the time specified by the deadline (i.e. five nano-seconds). A task in a real-time system must be

completed neither too early nor too late. A system is said to be unschedulable when tasks cannot meet the specified deadlines. A task can be classified as either a periodic or a periodic process.

The criteria of a real-time can be classified as hard, firm or soft. The scheduler set the algorithms for executing tasks according to a specified order. There are multiple mathematical models to represent a scheduling System; most implementations of real-time scheduling algorithm are modeled for the implementation of uniprocessors or multiprocessors configurations. In the algorithms for a real-time scheduling system, each task is assigned a description, deadline and an identifier (indicating priority). The selected scheduling algorithm determines how priorities are assigned to a particular task. A real-time scheduling algorithm can be classified as static or dynamic. For a static scheduler, task priorities are determined before the system runs. A dynamic scheduler determines task priorities as it runs. Tasks are accepted by the hardware elements in a real-time scheduling system from the computing environment and processed in real-time. An output signal indicates the processing status. A task deadline indicates the time set to complete for each task.

a. Priority Fair share scheduling

Fair share scheduling is a scheduling algorithm for computer operating systems in which the CPU usage is equally distributed among system users or groups as opposed to equal distribution among processes.

One common method of logically implementing the fair-share scheduling strategy is to recursively apply for the round-robin scheduling strategy at each level of abstraction (processes, users, groups, etc). The time quantum required by round-robin is arbitrary, as any equal division of time will produce the same results.

Example

If four users (A, B, C, D) are concurrently executing one process each, the scheduler will logically divide the available CPU cycles such that each user gets 25% of the whole (100% /4 =25%). If user B starts a second process, each user will still receive 25% of the total cycles, but each of user B's processes will now be attributed 12.5% of the total CPU cycles each, totaling user B's fair share of 25%. On the other hand, if a new user starts a process on the system, the scheduler will reapportion the available CPU cycles such that each user gets 20% of the whole (100% /5=20%).

Another layer of abstraction allows us to partition users into groups, and apply the fair share algorithm to the groups as well. In this case, the available CPU cycles are divided first among the groups, then among the users within the groups, and then among the processes for that user. For example, if there are three groups (1,2,3) containing three, two and four users respectively, the available CPU cycles will be distributed as follows:

100% /3 groups = 33.3% per group

Group 1: (33.3% /3 users) =11.1% per user

Group 2: (33.3% /2 users) = 16.7% per user

Group 3: (33.3% /4 users) = 8.3% per user

b. Guaranteed Scheduling

A scheduling algorithm used in multitasking operating systems that guarantees fairness by monitoring the amount of CPU time spent by each user and allocating resources accordingly.

It makes real promises to the users about performance. If there are n users logged in while you are working, you will receive about 1/n of the CPU power. Similarly, on a single-user system with n processes running, all things being equal, each one should get 1 /n of the CPU cycles. To make good on this promise, the system must keep track of how much CPU each process has had since its creation. CPU time consumed to the CPU time entitled. A ratio of 0.5 means that a process has only had half of what it should have had, and a ration of 2.0 means that a process has had twice as much as it was entitled to. The algorithm is then to run the process with the lowest ratio until its ratio has moved above its closest competitor.

c. Lottery Scheduling

Lottery Scheduling is type of process scheduling, somewhat different from other scheduling. Processes are scheduled in a random manner. Lottery scheduling can be preemptive or non-preemptive. It also solves the problem of starvation. Giving each process at least one lottery ticket guarantees that it has non-zero probability of being selected at each scheduling operation.

In this scheduling every process has some tickets and scheduler picks a random ticket and process having that ticket is the winner and it is executed for a time slice and then another ticket is picked by the scheduler. These tickets represent the share of processes. A process having a higher number of tickets give it more chance to get chosen for execution.

Example

If we have two processes A and B having 60 and 40 tickets respectively out of total 100 tickets. CPU share of A is 60% and that of B is 40%. These shares are calculated probabilistically and not deterministically.

Explanation:

1. We have two processes A and B. A has 60 tickets (ticket number 1 to 60) and B have 40 tickets (ticket no 61 to 100).
2. Scheduler picks a random number from 1 to 100. If the picked no. is from 1 to 60 then A is executed otherwise B is executed.
3. An example of 10 tickets picked by Scheduler may look like this-
   Ticket number: {73 82 23 45 32 87 49 39 12 09}
   Resulting schedule: {B B A A A B A A A A}

4. A is executed 7 times and B is executed 3 times. As you can see that A takes 70% of CPU and B takes 30% of CPU which is not the same as what we need as we need A to have 60% of CPU and B should have 40% of CPU. This happens because shares are calculated probabilistically but in a long run (i.e. when no. of tickets picked is more than 100 or 1000) we can achieve a share percentage of approx.. 60 and 40 for A and B respectively.

d. Highest Response Ratio Next (HRN)

HRN is non-preemptive scheduling algorithm. In Shortest Job First scheduling, priority is given to shortest job, which may sometimes indefinite blocking of longer job. HRN Scheduling is used to correct this disadvantage of SJF. For determining priority, not only the job's service time but the waiting time is also considered. In this algorithm, dynamic priorities are used instead of fixed priorities. Dynamic priorities in HRN are calculated as,

Priority = (waiting time + service time) /service time.

So, shortest job get preference over longer processes because service time appears in the denominator. Longer jobs that have been waiting for long period are also give favorable treatment because waiting time is considered in numerator.

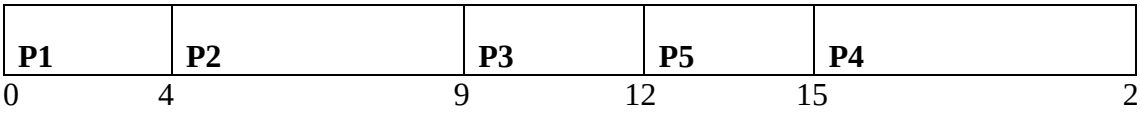
Steps of HRN implementation

1. Input the number of processes, their arrival time and burst times.
2. Sort them according to their arrival time.
3. At any given time calculate the response ratios and select the appropriate process to be scheduled.
4. Calculate the turnaround time as
   Turnaround = completion time – arrival time.
5. Calculate the waiting time as
   Waiting time = turnaround time – burst time
6. Turnaround time divided by the burst time gives the normalized turnaround time.
7. Sum up the waiting and turnaround times of all processes and divide by the number of processes to get the average waiting and turnaround time.

Example:

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

Solution



Explanation

At time t=0, only the process P1 is available in the ready queue. So, process P1 executes till its completion. At time t=4, only the process P2 and P3 are available in the ready queue. So, we have to calculate the response ratio. The process which has the highest response ratio will be executed next.

Response Ratio is,

Response Ratio of P2 = [(4-2)+5]/5 = 1.40

Response Ratio of P3 = [(4-4)+3]/3 = 1

Process P2 has highest response ratio so it will be selected for execution. After the completion of execution process P2, there are three processes P3, P4 and P5 are in the ready queue.

So, the Response Ratio for processes P3, P4 and P5 are,

Response Ratio of P3 = [(9-4)+3]/3 = 2.66

Response Ratio of P4 = [(9-6)+6]/6 = 1.50

Response Ratio of P5 = [(9-8)+3]/3 = 1.33

Process P3 has highest response ratio so it will be selected for execution. After the completion of execution of process P3, there are three processes P4 and P5 are in the ready queue. So, the Response Ratio for processes P4 and P5 are,

Response Ratio of P4 = [(12-6)+6]/6 = 2

Response Ratio of P5 = [(12-8)+3]/3 = 2.33

Process P5 has highest response ratio so it will be executed next. After the completion of the execution of process P5, there are only process P4 in the ready queue. So, it will be executed next.

Advantages

- Its performance is better than SJF scheduling.
- It limits the waiting time of longer jobs and also supports shorter jobs.

Disadvantages

- It can't be implemented practically.
- This is because the burst time of all the processes cannot be known in advance.

Exercise 1

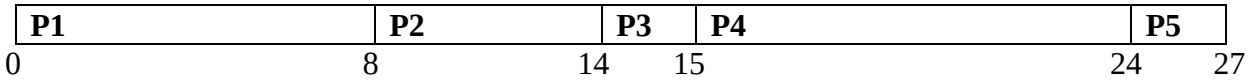
For the process listed in following table, draw Gantt chart illustrating their execution and calculate average waiting time and turnaround time using:

a. First Come First Served
b. Shortest Job First
c. Priority
d. Round Robin (2ms Quantum)

Process	Burst	Priority
P1	8	4
P2	6	1
P3	1	2
P4	9	2
P5	3	3

Solution:

Gantt chart for First Come First Serve Scheduling



Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	8-0 = 8	8-8 = 0
P2	14-0 = 14	14-6 = 8
P3	15-0 = 15	15-1 = 14
P4	24-0 = 24	24-9 = 15
P5	27-0 = 27	27-3 = 24

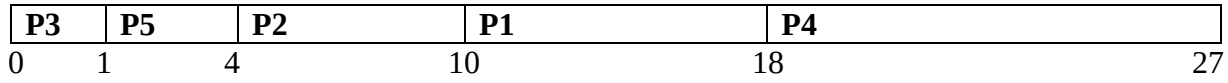
Total waiting time: (0+8+14+15+24) = 61 ms

Average waiting time: 61/5 = 12.2 ms

Total turnaround time: (8+14+15+24+27) = 88 ms

Average turnaround time: 88/5 = 17.6 ms

Gantt chart for Shortest Job First Scheduling



Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	18-0 = 18	18-8 = 10
P2	10-0 = 10	10-6 = 4
P3	1-0 = 1	1-1 = 0
P4	27-0 = 27	27-9 = 18
P5	4-0 = 4	4-3 = 1

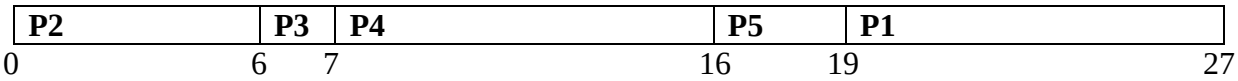
Total waiting time: (10+4+0+18+1) = 33 ms

Average waiting time: 33/5 = 6.6 ms

Total turnaround time: (18+10+1+27+4) = 60 ms

Average turnaround time: 60/5 = 12 ms

Gantt chart for Priority Scheduling



Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	27-0 = 27	27-8 = 19
P2	6-0 = 6	6-6 = 0
P3	7-0 = 7	7-1 = 6
P4	16-0 = 16	16-9 = 7
P5	19-0 = 19	19-3 = 16

Total waiting time: (19+0+6+7+16) = 48 ms

Average waiting time: 48/5 = 9.6 ms

Total turnaround time: (27+6+7+16+19) = 75 ms

Average turnaround time: 75/5 = 15 ms

Gantt chart for Round Robin Scheduling (2ms Quantum)

P1	**P2**	**P3**	**P4**	**P5**	**P1**	**P2**	**P4**	**P5**	**P1**	**P2**	**P4**	**P1**	**P4**

0 2 4 5 7 9 11 13 15 16 18 20 22 24 27

Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	24-0 = 24	24-8 = 16
P2	20-0 = 20	20-6 = 14
P3	5-0 = 5	5-1 = 4
P4	27-0 = 27	27-9 = 18
P5	16-0 = 16	16-3 = 13

Total waiting time: (16+14+4+18+13) = 65 ms

Average waiting time: 65/5 = 13 ms

Total turnaround time: (24+20+5+27+16) = 92 ms

Average turnaround time: 92/5 = 18.4 ms

Exercise 2

For the process listed in following table, draw Gantt chart illustrating their execution and calculate average waiting time and turnaround time using:

a. First Come First Served
b. Shortest Job First
c. Priority
d. Round Robin (2ms Quantum)

Process	Burst	Priority	Arrival Time
P1	8	4	0
P2	6	1	2
P3	1	2	2
P4	9	2	1
P5	3	3	3

Solution:

Gantt chart for First Come First Served Scheduling

Ready Queue	**P1**	**P5P3P2P4**

0 8

Running Queue	**P1**	**P4**	**P2**	**P3**	**P5**

0 8 17 23 24 27

Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	8-0 = 8	8-8 = 0
P2	23-2 = 21	21-6 = 15
P3	24-2 = 22	22-1 = 21
P4	17-1 = 16	16-9 = 7
P5	27-3 = 24	24-3 = 21

Total waiting time: (0+15+21+7+21) = 64 ms

Average waiting time: 64/5 = 12.8 ms

Total turnaround time: (8+21+22+16+24) = 91 ms

Average turnaround time: 91/5 = 18.2 ms

Gantt chart for Shortest Job First

P1	**P3**	**P5**	**P2**	**P1**	**P4**

0 2 3 6 12 18 27

Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	18-0 = 18	18-8 = 10
P2	12-2 = 10	10-6 = 4
P3	3-2 = 1	1-1 = 0
P4	27-1 = 26	26-9 = 17
P5	6-3 = 3	3-3 = 0

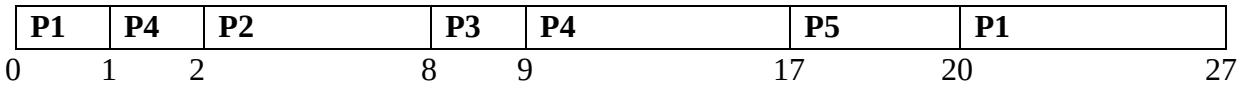
Total waiting time: (10+4+0+17+0) = 31 ms

Average waiting time: 31/5 = 6.2 ms

Total turnaround time: (18+10+1+26+3) =58 ms

Average turnaround time: 58/5 = 11.6 ms

Gantt chart for Priority Scheduling



Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	27-0 = 27	27-8 = 19
P2	8-2 = 6	6-6 = 0
P3	9-2 = 7	7-1 = 6
P4	17-1 = 16	16-9 = 7
P5	20-3 = 17	17-3 = 14

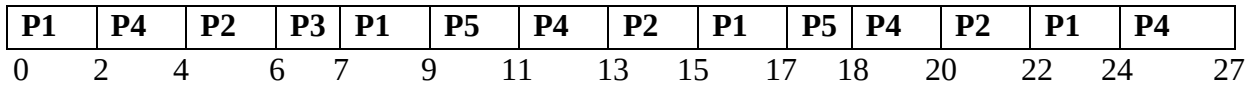
Total waiting time: (19+0+6+7+14) = 46 ms

Average waiting time: 46/5 = 9.2 ms

Total turnaround time: (27+6+7+16+17) =73 ms

Average turnaround time: 73/5 = 14.6 ms

Gantt chart for Round Robin (2ms Quantam)



Process	Turnaround time = Completion Time- Arrival Time	Waiting time = Turn Around Time – Burst Time
P1	24-0 = 24	24-8 = 16
P2	22-2 = 20	20-6 = 14
P3	7-2 = 5	5-1 = 4
P4	27-1 = 26	26-9 = 17
P5	18-3 = 15	15-3 = 12

Total waiting time: (16+14+4+17+12) = 63 ms

Average waiting time: 63/5 = 12.6 ms

Total turnaround time: (24+20+5+26+15) =90 ms

Average turnaround time: 90/5 = 18 ms