PROCESS CONCEPT

A program under execution is known as Process. A process is a active entity & is also known as job.

Process:

A program consists of:

- Text section: All the instruction of the process is stored in this section.
- Stack: It consists of temporary data such as function parameters, return addresses & local variables.
- Data section: contains global variables.
- Heap: is memory that is dynamically allocated during process run-time.

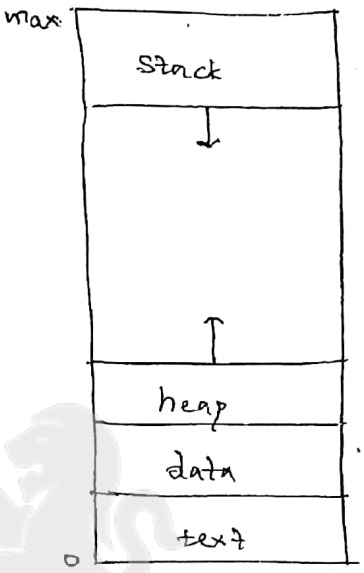


Fig: Process in memory.

There can be situation where in which two processes are associated with a same program.

Ex: several users may be running different copies of the mail program. or the same user may invoke many copies of the web browser program. Each of these is a separate process & text sections are equivalent, the data, heap, & stack section vary.

Process state

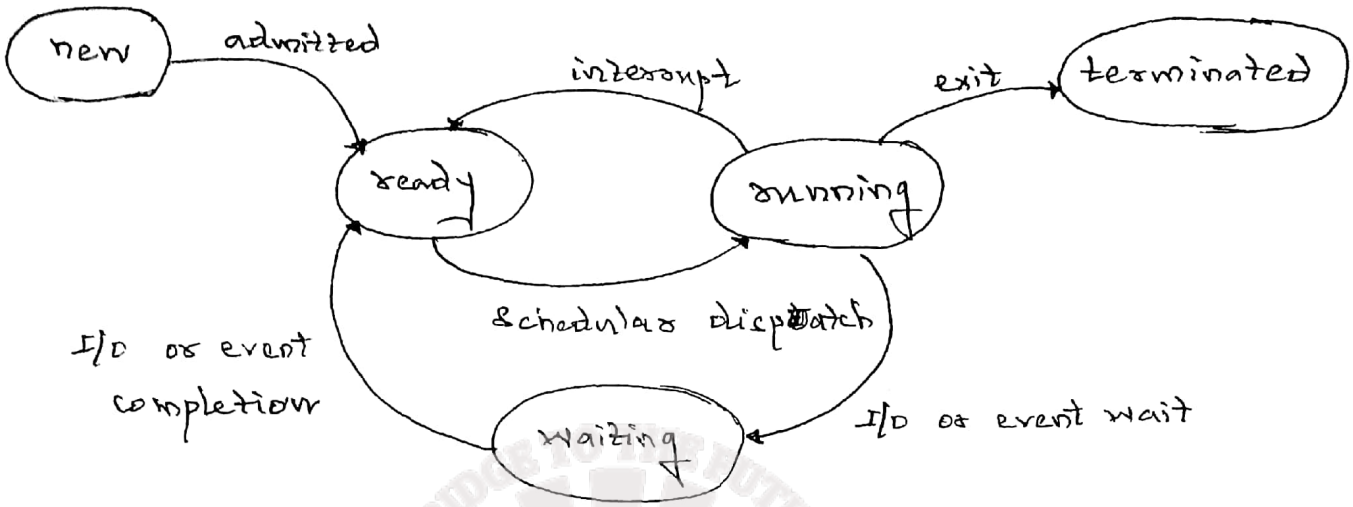


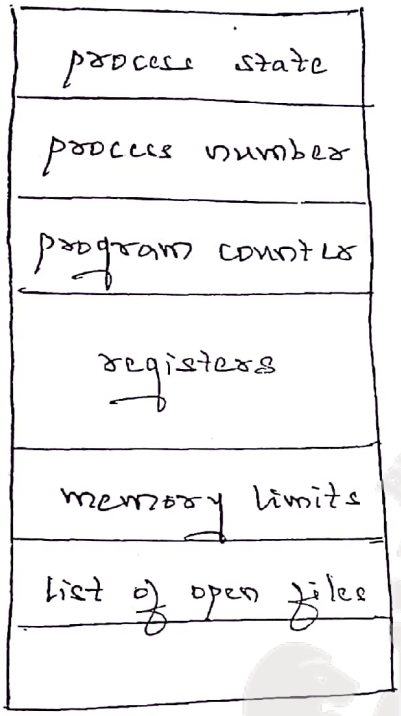
Fig: Diagram of process state.

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.

Each process may be in one of the following states.

- **New**: The process is being created
- **Running**: Instructions are being executed
- **Waiting**: The process is waiting for some event to occur such as an I/O completion or reception of a signal.
- **Ready**: The process is waiting to be assigned to a processor
- **Terminated**: The process has finished execution.

Process Control Block: (PCB)



↳ each process is represented in the OS by a process control block.

↳ also called a task control block.

Ans: Process Control Block.

It contains many pieces of information associated with a specific process, including these:

- Process state: The state may be new, ready, running, waiting, halted & so on.
- Program counter: The counter indicates the address of the next instructions to be executed for this process.
- Process number: Is a unique number associated with each process & also known as process identifier (PID).
- CPU registers: This includes accumulators, index registers, stack pointers & general-purpose registers.

- CPU-scheduling information: This information includes a process priority, pointers to scheduling queues & any other scheduling parameters.
- Memory-management information: Includes information as the value of the base & limit registers, the page tables, or the segment tables, depending on the memory system used by the OS.
- Accounting information: This includes the amount of CPU & real time used, time limits, account numbers, job or process numbers & so on.
- I/O status information: This includes the list of I/O devices allocated to the process, a list of open files & so on.

Process Scheduling:

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among process so frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process for program execution on the CPU.

Scheduling Queues:

Process scheduling is achieved by using different types of scheduling queues.

1. Job Queue.
2. Ready Queue.
3. Device Queue.
4. I/O Queue.

Job Queue: consists of all processes in the system.

Ready Queue: The processes that are residing in main memory & are ready & waiting to execute are kept on a list called the ready queue.

The list of processes waiting for a particular I/O device is called **device queue**.

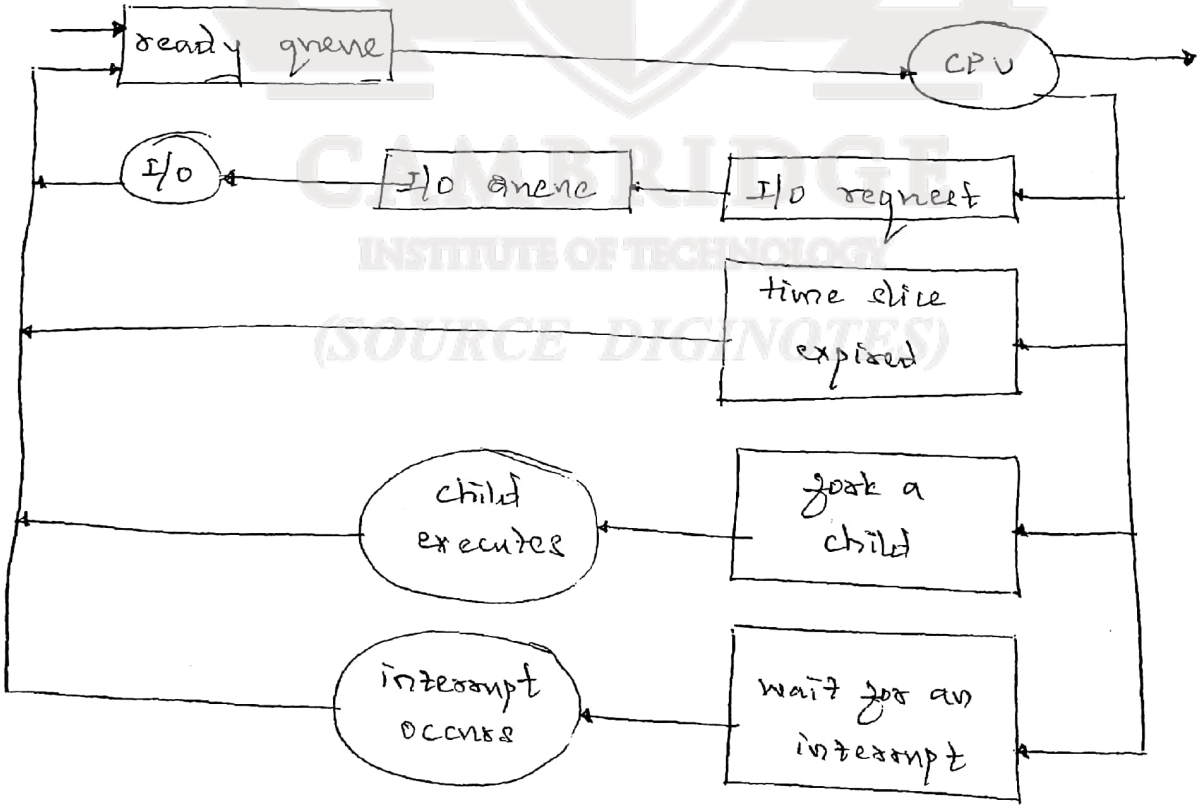


Fig: Queueing - diagram representation of process

A new process is initially put in the ready queue. It waits there until it is selected for execution or is dispatched. Once the process is allocated the CPU and is executing, one of several events could occur.

- the process could issue an I/O request and then be placed in an I/O queue.
- the process could create a new subprocess & wait for the subprocess termination.
- the process could be removed forcibly from the CPU, as a result of an interrupt & be put back in the ready queue.

Schedulers

- long-term scheduler or job scheduler
- short-term scheduler or CPU scheduler.

Context switch

When the CPU gets interrupted, the O.S must save the current state or information of the process which is under execution so that execution of the process can be resumed after interrupt is serviced.

Switching the CPU to another process requires performing a state save of the current process & a state restore of a different process. This task is known as a context switch.

When a context switch occurs, the kernel saves the context of the old process in its PCB & loads the saved context of the new process scheduled to run.

Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from m/c to m/c, depending on the memory speed, no. of registers that must be copied & existence of special instructions.

Context-switch times are highly dependent on hardware support.

Operations on Processes

The processes in most systems can execute concurrently & they can be created & deleted dynamically.

The main operation on the process are:

1. Process Creation
2. Process Termination

A process may create several new processes via a create-process system call, during the course of execution. The creating process is called a parent process & the new processes are called the children of that process.

Most OS identify process according to a unique process identifier (pid) which is typically an integer number.

Each of these new processes may in turn create other processes, forming a tree of processes.

A process will need certain resources to accomplish its task. When a process creates a subprocess, subprocess may be able to obtain its resources directly from the OS or it may be constrained to a subset of the resources of the parent process.

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

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

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

1. The child process is a duplicate of the parent process. (It has same program & data as the parent).
2. The child process has a new program loaded into it.

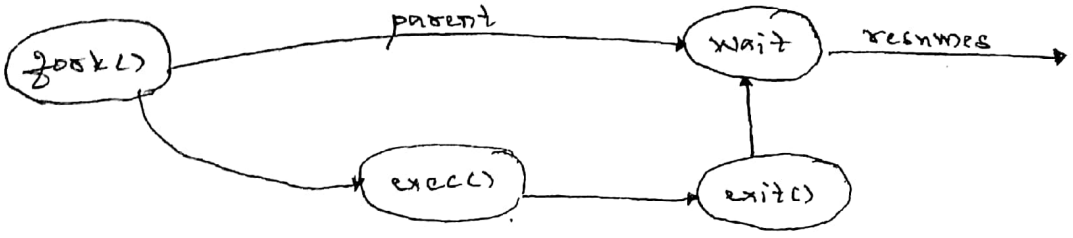


fig: Process creation using fork() system call.

Creating a separate process using the UNIX fork() system call

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>

int main()
{
    pid_t pid;
        /* fork a child process */
    pid = fork();
    if (pid < 0) {      /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) {        /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else {  /* parent process will wait for the child to
                                                complete */
        wait(NULL);
        printf("child complete");
    }
    return 0;
}
```

Process Termination:

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

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

- the child has exceeded its usage of some of the resources that it has been allocated.
- the task assigned to the child is no longer required.
- the parent is exiting, the OS doesn't allow a child to continue if its parent terminates.

Interprocess communication: (IPC).

Processes executing in the system can be grouped as

- Independent processes
- co-operating processes.

A process is said to be independent process if it is not affecting any other process or get affected by other process.

Any process that doesn't share data with any other process is independent.

A process is cooperating if it can affect or be affected by the other processes executing in the system.

There are several reasons for providing an environment that allows process cooperation:

1. Information sharing: Since several users may be interested in the same piece of information (eg: file sharing). it must provide an environment to allow concurrent access to such information.

2. Computation speedup: A particular task to run faster, it will be broken into subtasks & executed in parallel, such a speed up can be achieved only if the computer has multiple processing elements.

3. Modularity: Systems are constructed in a modular fashion, dividing the system functions into separate process or threads.

4. Convenience: Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing & compiling in parallel.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data & information. There are two models of IPC,

1. Shared memory.
2. Message passing.

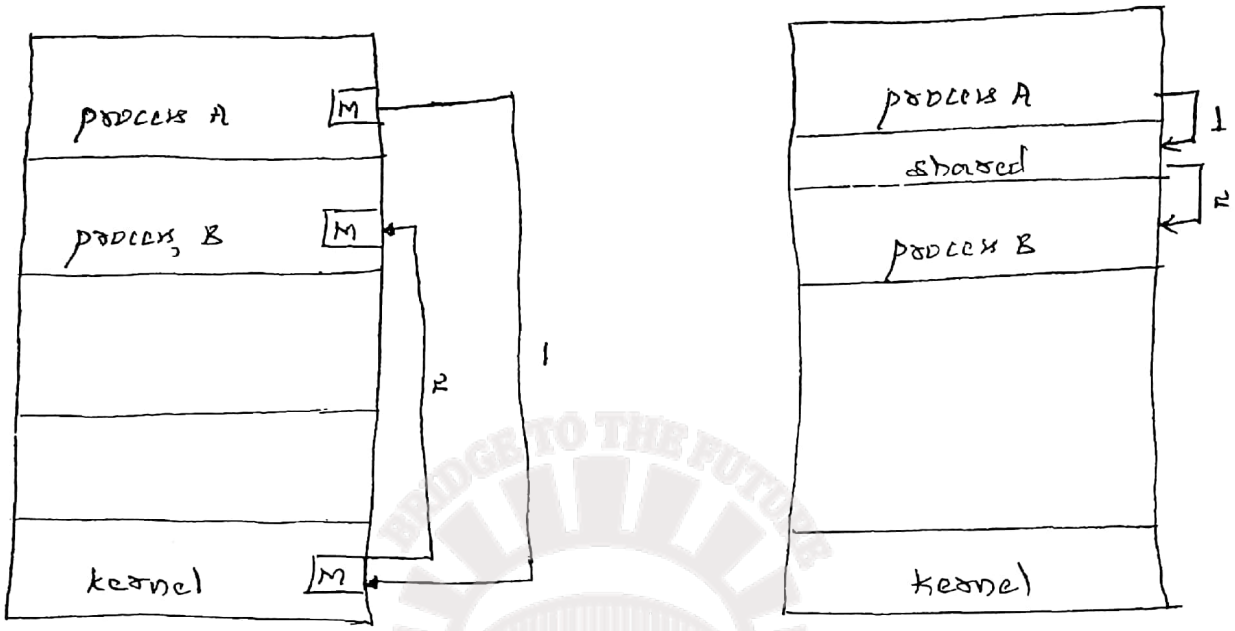


Fig (a). Message passing. (b) Shared memory.

In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading & writing data to the shared region.

In Message passing model, communication takes place by means of messages exchanged b/w the cooperating processes.

Message Passing	Shared Memory
1. useful for exchanging smaller amount of data.	Larger amount of data
2. Easier to implement	difficult to implement
3. slower in speed	allows maximum speed of communication.
4. Implemented using system calls & thus require the more time-consuming task of kernel intervention.	system calls are required only to establish shared memory regions. No assistance from the kernel is required.

Shared-Memory systems:

To illustrate the concept of cooperating processes, consider the producer-consumer problem. A producer process produces information that is consumed by a consumer process.

For example, a compiler may produce assembly code, which is consumed by an assembler, the assembler, in turn, may produce object modules, which are consumed by the loader.

One solution to the producer-consumer problem uses shared memory. To allow producer & consumer processes to run concurrently, a buffer is need, a buffer is a collection of items that can be filled by the producer & emptied by the consumer. This buffer will reside in a region of memory that is shared by producer & consumer process.

Two types of buffers can be used:

1. The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, & the producer must wait if the buffer is full.

2. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.

Shared buffer:

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared buffer is implemented as a circular array with two logical pointers: in & out. The variable in points to the next free position in the buffer, out points to the first full position in the buffer.

The buffer is empty when in == out;
the buffer is full when ((in+1) % BUFFER_SIZE) == out.

The producer process:

```
item nextProduced;
while (true) {
    /* produce an item in nextProduced */
    while ((in+1) % BUFFER_SIZE) == out)
        ;   /* do nothing */
    buffer[in] = nextProduced;
    in = (in+1) % BUFFER_SIZE;
}
```

The consumer process:

```
item nextConsumed;
while (true) {
    while (in == out)
        ;   // do nothing
    nextConsumed = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    /* consume the item in nextConsumed */
}
```

The code for the producer & consumer process is as given above. The producer process has a local variable nextProduced in which new item to be produced is stored. The consumer process has a local variable nextConsumed in which item to be consumed is stored.

Message-Passing systems.

Message passing provides a mechanism to allow processes to communicate & to synchronize their actions without sharing the same address space & is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

Ex: a chat program used on the world wide web could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations: send & receive (message). Message sent by a process can be of either fixed or variable size.

If processes P & Q want to communicate, they must send messages to and receive messages from each other, a communication link must exist b/w them.

Link can be implemented in a variety of ways:

→ Direct or indirect communication.

→ Synchronous or asynchronous communication.

→ Automatic or explicit buffering.

Naming:

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Direct communication:

In this scheme, each process that want to communicate must explicitly name the recipient or sender of the communication.

The send() & receive() primitives are defined as:

- send (P, message) - send a message to process P.
- receive (Q, message) - Receive a message from process Q

A communication link in this scheme has the following properties:

- A link is established automatically b/w every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of process, there exists exactly one link.

Symmetric addressing

↳ both the sender process & the receiver process must name the other to communicate.

Asymmetric addressing:

↳ Here, only the sender names the recipient, the recipient is not required to name the sender.

In asymmetric addressing, the send() & receive() primitives are defined as follows.

- send(P, message) - send a message to process P.
- receive(id, message) - Receive a message from any process, the variable id is set to the name of the process with which communication has taken place.

Disadvantage: (in both schemes)

↳ is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions.

Indirect communication:

The messages are sent to & received from mailboxes or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes & from which messages can be removed. Each mailbox has a unique identification.

In this scheme, a process can communicate with some other process via a no. of diff. mailboxes. Two processes can communicate only if the processes have a shared mailbox.

The send() & receive() primitives are defined as:

- send(A, message) - send a message to mailbox A.
- receive(A, message) - Receive a message from mailbox A.

A communication link has the foll. properties:

- A link is established b/w a pair of process only if both members of pair have a shared mailbox.
- A link may be associated with more than two processes
- Between each pair of communicating processes, there may be a no. of different links, with each link corresponding to one mailbox.

Now suppose that processes P_1, P_2 & P_3 all share mailbox A. Process P_1 sends a message to A, while both P_2 & P_3 execute a receive() from A. which process will receive the message sent by P_1? the answer depends on which of the foll. methods chosen:

- Allow a link to be associated with two processes at most.
- Allow at most one process at a time to execute a receive() operation.
- Allow the system to select arbitrarily which process will receive the message.

A mailbox may be owned either by a process or by the operating system.

If the mailbox is owned by a process, then owner & users are distinguished. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that mailbox no longer exists.

A mailbox owned by the operating system has an existence of its own. It is independent & is not attached to any particular process. The OS then must provide a mechanism that allows a process to do the following:

- Create a new mailbox
- Send & receive messages through the mailbox
- Delete a mailbox.

Synchronization:

Communication b/w processes takes place through calls to send() & receive() primitives. There are different design options implementing each primitive.

Message passing may be either blocking or nonblocking Also known as synchronous & asynchronous.

- Blocking send: The sending process is blocked until the message is received by the receiving proces or by the mailbox.
- Nonblocking send: The sending process sends message & resumes its operations.
- Blocking receive: The receiver is blocked until a message is available.
- Nonblocking receive: The receiver retrieves either a valid message or null.

Synchronous: → both send & receives primitive are blocking.

Asynchronous: → one of them or both are nonblocking.

Buffering:

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Such queues can be implemented in three ways:

1. Zero capacity: The queue has a maximum length of zero. Thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

2. Bounded capacity: The queue has finite length n, thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue & the sender can continue execution without waiting. The link capacity is finite. If the link is full, the sender must block until space is available in the queue.

3. Unbounded capacity: The queue's length is potentially infinit Thus, any no. of messages can wait in it. The senders never blocks.

thread

↳ light-weight process (LWP).

Multithreaded Programming

A thread is a basic unit of CPU utilization, it comprises a thread ID, a program counter, a registers set & a stack.

Motivation:

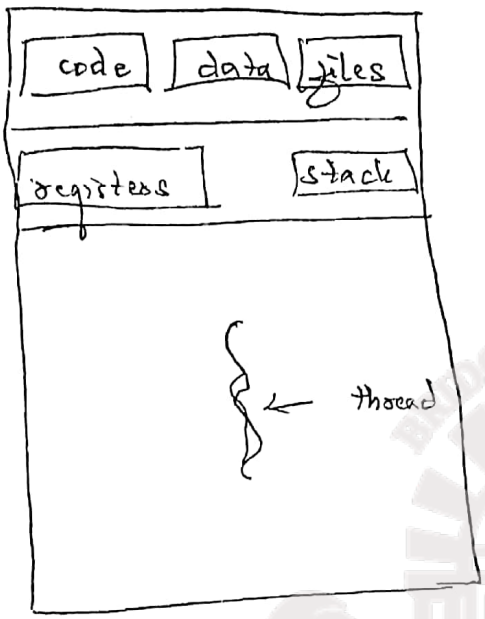
An application typically is implement as a separate process with several threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network.

In certain situations, a single application may be required to perform several similar tasks. Ex: A web server accepts client request for webpage, image, sound & so on. It may have several clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time & a client might have to wait a very long time.

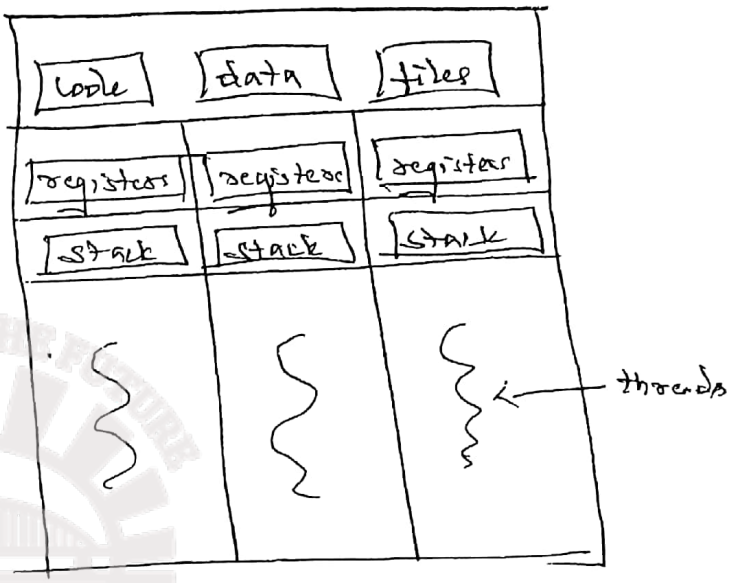
One solution is to use process-creation method. process creation is time consuming & resource intensive.

The better approach would multithread the web server process. Thus threads gain considerable performance.

- the server would create a separate thread that would listen for client requests.
- when a request was made, rather than creating another process, the server would create another thread to service the request.



(i) Single-threaded process.



(ii) multithreaded process.

Benefits:

1. Responsiveness.
2. Resource sharing.
3. Economy.
4. Scalability.

Multithreading Models:

Support for threads may be provided either at the user level for user threads or by the kernel, for kernel threads.

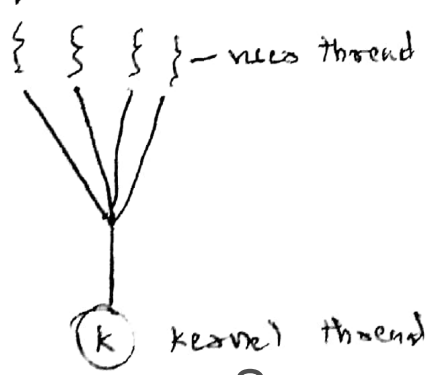
↳ user threads are supported above the kernel & are managed without kernel support.

↳ kernel threads are supported & managed directly by OS.

Three common ways of establishing relationships b/w kernel & user threads.

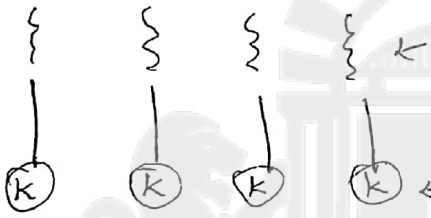
1. Many to One Model.
2. One to One Model.
3. Many to Many Model.

Many to One Model:



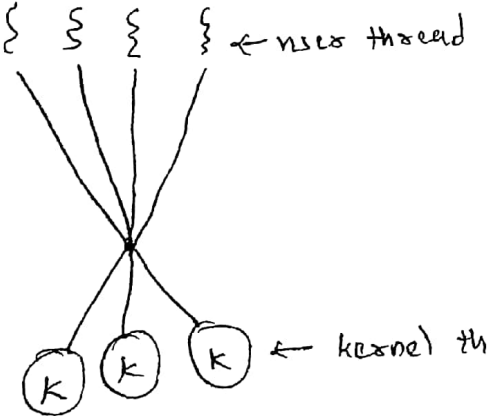
- maps many user-level threads to one kernel thread.
- the entire process will block if a thread makes a blocking system call.
- only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.

One to one Model:



- maps each user thread to a kernel thread.
- It provides more concurrency than the many-to-one.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Linux, along with family of Windows OSs implement this model.

Many to Many Model:



- multiplexes many user-level threads to a smaller or equal no. of kernel threads.
- the no. of kernel threads may be specific to either a particular application or a particular machine.
- It allows the developer to create as many user threads as they wish, true concurrency is not gained because the kernel can schedule only one thread at a time.

The many-to-many model suffers from neither of these shortcomings

1. Developers can create as many user threads as necessary & the corresponding kernel threads can run in parallel on a multiprocessor.
2. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

Thread libraries.

A thread library provides the programmer an API for creating & managing threads. There are two ways of implementing a thread library.

1. The first approach is to provide a library entirely in user space with no kernel support. All code & data structures for library exist in user space. This means that invoking a function in the library results in a local function call in user space & not a system call.

2. The second approach is to implement a kernel-level library supported directly by OS. The code & data structures for the library exist in kernel space. Invoking a function in the library results in a system call to the kernel.

Three main thread libraries are:

1. POSIX Pthreads: the threads extension of the POSIX standard, may be provided as either a user - or kernel-level library.

2. Win32: Is a kernel-level library available on windows systems.

3. Java: The java thread API allows thread creation & management directly in java programs. However, because in most instance the JVM is running on top of a host os, the java thread API is typically implemented using a thread library available on the host system.

Threading Issues:

- The fork() & exec() system calls
- Cancellation
- Signal Handling
- Thread Pools
- Thread-specific Data
- Scheduler Activations

The fork() & exec() system calls

- If one thread in a program calls fork(),
  - does the new process duplicate all threads
  - or is the new process single-threaded?
- some UNIX systems have chosen to have two versions of fork().
  - one that duplicates all threads
  - and another that duplicates only the thread that invoked the fork() system call.
- which of the two versions of fork() to use depends on the application.
  - If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the prog. specified in the parameters to exec() will replace the process. In this instance, duplicating only the calling thread is appropriate.
  - If however, the separate process does not call exec() after forking, the separate process should duplicate all threads.

Cancellation:

Thread cancellation is the task of terminating a thread before it has completed.

Eg:
- If multiple threads are concurrently searching through a database & one thread returns the result, the remaining threads might be canceled.
- Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further.

A thread that is to be canceled is often referred to as the target thread. Cancellation of a target thread may occur in two different scenarios:

1. Asynchronous cancellation: One thread immediately terminates the target thread.
   → The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads.
   → Often, the OS will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resources.

2. Deferred cancellation: The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.
   - With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine if it should be cancelled or not.
   - This allows a thread to check whether it should be canceled at a point when it can be canceled safely. Pthreads refers to such points as cancellation points

Signal handling

- A signal is used in UNIX systems to notify a process that particular event has occured.
- A signal may be received either synchronously or asynchronously depending on the source of & the reason for the event being signaled.
- All the signals, follow the same pattern:
  1. A signal is generated by the occurrence of a particular event.
  2. A generated signal is delivered to a process.
  3. Once delivered, the signal must be handled.
- Examples of synchronous signals include illegal memory access & division by 0. If a running prog. performs either of these action, a signal is generated.
- synchronous signals are delivered to the same process that performed the operation that caused the signal. (i.e. the reason they are considered synchronous)
- When a signal is generated by an event external to a running process, that process receives the signal asynchronously.
- Examples of such signals include terminating a process with specific keystrokes such as <control><c> & having a timer expire.
- Typically, ~~signal may be handled by~~ an asynchronous signal is sent to another process.

A signal may be handled by one of two possible handlers:

1. A default signal handler.
2. A user-defined signal handler.

Every signal has a default signal handler that is run by the kernel when handling that signal. This default action can be overriden by a user-defined signal handler that is called to handle the signal.

Thread Pools:

Issue in multithreading are:

1. The amount of time required to create the thread prior to servicing the request, together with the fact that thread will be discarded once it has completed its work.
2. Not placed a bound on the no. of threads concurrently active in system. Unlimited threads could exhaust system resources such as CPU time or memory.

One solution to this problem is to use a thread pool.

The general idea behind a thread pool is to create a no. of threads at process startup and place them into a pool, where they sit & wait for work.

When a server receives a request, it awakens a thread from this pool - if one is available & passes it the request for service. If the pool contains no available threads, the server waits until one becomes free.

Benefits

1. Servicing a request with an existing thread is usually faster than waiting to create a thread.
2. A thread pool limits the no. of threads that exist at any point.

Thread specific data

A data which is specific to a particular thread is called as thread specific data

Ex: In a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specific data.

Schedules Activations

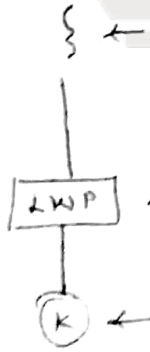


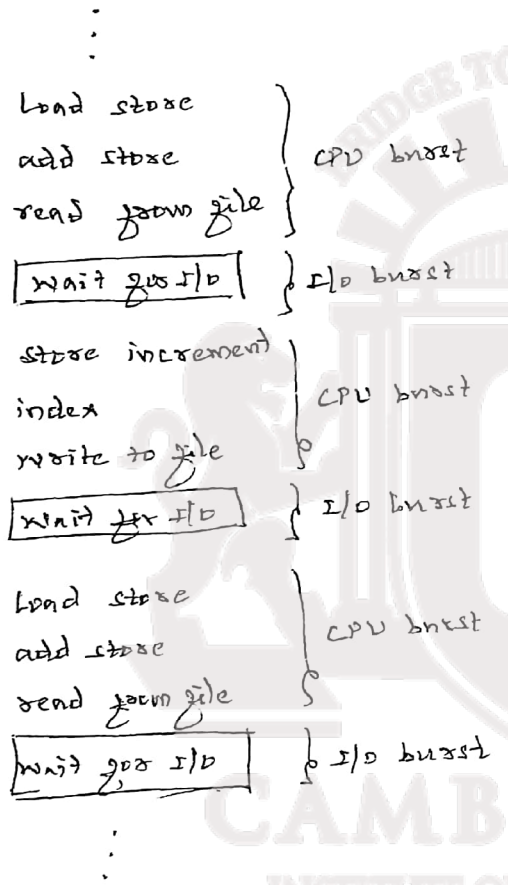
Fig: Lightweight process (LWP)

PROCESS SCHEDULING

CPU scheduling is the basis of multiprogrammed OS. By switching the CPU among processes, the OS can make the computer more productive.

CPU-I/O Burst cycle:

The execution of process consists of a cycle of CPU execution & I/O wait. Process alternate b/w these two states.



Process execution begins with a CPU burst, followed by an I/O burst, which is followed by another CPU burst, the another I/O burst & so on.

Eventually, the final CPU burst ends with a system request to terminate execution.

Fig: Alternating sequence of CPU & I/O bursts.

CPU schedulers:

Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed. The selection process is carried out by the CPU scheduler or short-term scheduler. The scheduler selects a process from the processes in memory that are ready to execute & allocates the CPU to that process.

Dispatcher:

The dispatcher is the module that gives control of the CPU to the process 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.

The time it takes for the dispatcher to stop one process & start another running is known as the dispatch latency.

Preemptive scheduling

In preemptive scheduling, the server can be switched to the processing of a new request before completing the current request. The preempted request is put back into the list of pending requests. Its servicing would be resumed when it is scheduled again. Thus, a request may have to be scheduled many times before it completes.

The preemptive scheduling used in multiprogramming & time-sharing operating systems.

Non-preemptive scheduling

A server always processes a scheduled request to completion. Scheduling is performed only when processing of the previously scheduled request gets completed.

Non-preemptive scheduling is attractive due to its simplicity - It is not necessary to maintain a distinction b/w an unserviced request & a partially serviced one.

Scheduling Criteria.

Different CPU-scheduling algorithms have diff. properties. Many criteria have been suggested for comparing these algorithms.

The criteria include the following:

1. CPU utilization: To keep the CPU as busy as possible. CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent to 90 percent.

2. Throughput: The no. of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour, for short transactions, it may be ten processes per second.

3. Turnaround time: The interval from the time of submission of a process to the time of completion is the turnaround time. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on CPU & doing I/O.

4. Waiting time: Is the sum of the periods spent waiting in the ready queue.

5. Response time: It is the time from the submission of a request until the first response is produced.

Scheduling Algorithms

- First-come, First-served scheduling
- Shortest-Job-First scheduling
- Priority scheduling
- Round-Robin scheduling.

First-come, First-served scheduling (FCFS):

With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. The code for FCFS scheduling is simple to write & understand.

On the negative side, the average waiting time under the FCFS policy is often quite long.

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

Process	Burst Time
P_1	24
P_2	3
P_3	3

Turnaround time = Waiting time + Burst time

Gantt chart:

Waiting time of $P_1 = 0$
" $P_2 = 24$
" $P_3 = 27$

Avg. wt. time $= \frac{0+24+27}{3}$

$= \frac{51}{3}$

$= 17$ ms

Turnaround time of $P_1 = 24$
" $P_2 = 27$
" $P_3 = 30$

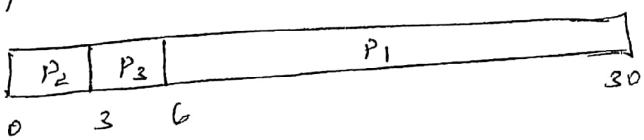
Avg. TAT $= \frac{24+27+30}{3}$

$= \frac{81}{3}$

$= 27$ ms

---

processes arrive in the order P_2, P_3, P_1



Waiting time of $P_1 = 6$
$P_2 = 0$
$P_3 = 3$

Avg. wt. time $= \frac{(6+0+3)}{3} = \frac{9}{3} = 3$ ms

TAT of $P_1 = 30$
$P_2 = 3$
$P_3 = 6$

Avg. TAT $= \frac{39}{3} = 13$ ms

convoy effect: all the other processes wait for one big process to get off the CPU.

The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

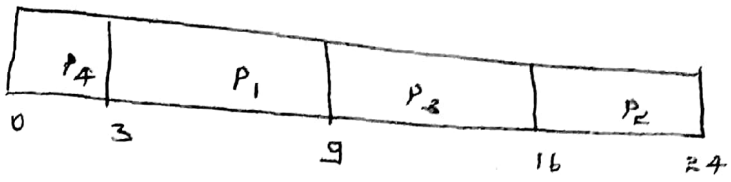
Shortest-Job-First Scheduling

This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the CPU bursts of two processes are the same, FCFS algorithm/scheduling is used to break the tie.

Example: Consider the following set of processes, with the length of the CPU burst given in milliseconds.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Gantt chart :



Waiting time of $P_1 = 3$

$P_2 = 16$

$P_3 = 9$

$P_4 = 0$

Thus, average waiting time $= \frac{(3+16+9+0)}{4}$

$= 28/4$

$= 7$ ms

TAT of $P_1 = 9$

$P_2 = 24$

$P_3 = 16$

$P_4 = 3$

Thus, average TAT $= \frac{52}{4}$

$= 13$ ms

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes.

Disadvantages of SJF:

1. The real difficulty with the SJF algorithm is knowing the length of the next CPU request.

2. It can't be implemented at the level of short-term CPU scheduling. With short-term scheduling, there is no way to know the length of the next CPU burst.

SJF → Preemptive SJF / SRTF - shortest remaining time first

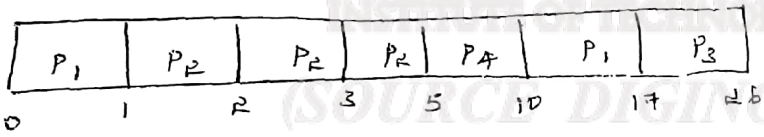
→ Non-preemptive.

A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.

Preemptive SJF / SRTF:

Process	Arrival time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Gantt chart:



Waiting Time of $P_1 = 10 - 1 = 9$

$P_2 = 0$

$P_3 = 17 - 2 = 15$

$P_4 = 5 - 3 = 2$

Average WT = $\frac{26}{4} = 6.5$ ms

TAT of $P_1 = 17$

$P_2 = 5 - 1 = 4$

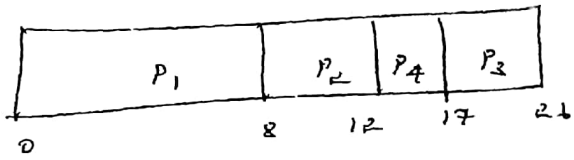
$P_3 = 24 - 2 = 24$

$P_4 = 10 - 3 = 7$

Avg. TAT = $\frac{52}{4} = 13$ ms

Non- Preemptive SJF

process	AT	BT
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



WT of $P_1 = 0$

$P_2 = 8 - 1 = 7$

$P_3 = 17 - 2 = 15$

$P_4 = 12 - 3 = 9$

Avg. WT $= \frac{31}{4} = 7.75$ ms

TAT of $P_1 = 8$

$P_2 = 12 - 1 = 11$

$P_3 = 26 - 2 = 24$

$P_4 = 17 - 3 = 14$

Avg. TAT $= \frac{57}{4} = 14.25$ ms

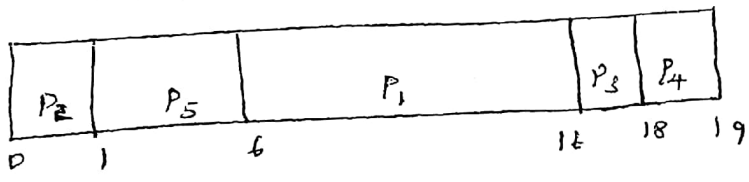
Priority scheduling:

A priority is associated with each process, & the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

Example:

process	BT	AT
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Gantt chart:



WT of $P_1 = 6$

$P_2 = 0$

$P_3 = 16$

$P_4 = 18$

$P_5 = 1$

Avg. WT $= \frac{41}{5} = 8.2$ ms

TAT of $P_1 = 16$

$P_2 = 1$

$P_3 = 18$

$P_4 = 19$

$P_5 = 6$

Avg. TAT $= \frac{60}{5} = 12$ ms.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.

A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

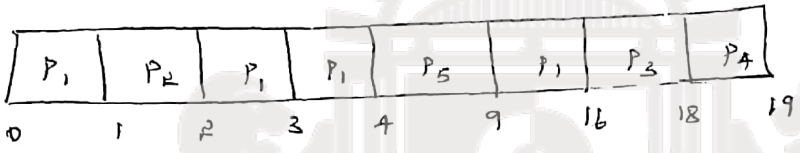
A major problem with priority scheduling algorithms is indefinite blocking or starvation. This algorithm can leave some low priority processes waiting indefinitely.

A solution to the problem is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

Preemptive Priority Scheduling

process	BT	Priority	AT
P_1	10	3	0
P_2	1	1	1
P_3	2	4	2
P_4	1	5	3
P_5	5	2	4

Gantt chart:



WT of $P_1 = 6$

$P_2 = 1 - 1 = 0$

$P_3 = 16 - 2 = 14$

$P_4 = 18 - 3 = 15$

$P_5 = 0.$

Avg. WT $= \frac{35}{5} = 7$ ms

TAT of $P_1 = 16$

$P_2 = 2 - 1 = 1$

$P_3 = 18 - 2 = 16$

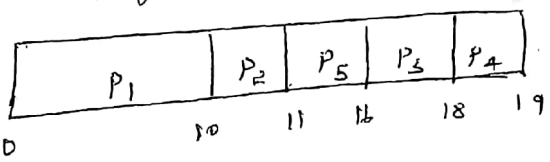
$P_4 = 19 - 3 = 16$

$P_5 = 9 - 4 = 5$

Avg. TAT $= \frac{54}{5} = 10.8$ ms

Non-Preemptive priority Scheduling:

Gantt chart:



WT of $P_1 = 0$

$P_2 = 10 - 1 = 9$

$P_3 = 16 - 2 = 14$

$P_4 = 18 - 3 = 15$

$P_5 = 11 - 4 = 7.$

Avg. WT $= 45/5 = 9$ ms

TAT of $P_1 = 10$

$P_2 = 11 - 1 = 10$

$P_3 = 18 - 2 = 16$

$P_4 = 19 - 3 = 16$

$P_5 = 16 - 4 = 12$

Avg. TAT $= \frac{64}{5} = 12.8$ ms

Round - Robin scheduling

It is designed especially for time sharing systems.
It is similar to FCFS scheduling, but preemption is added to enable the system to switch b/w processes.

* small quantum

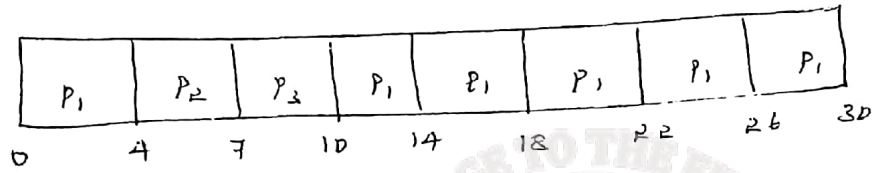
A small unit of time, called a time-quantum or time slice is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of upto 1 time quantum.

To implement RR scheduling, keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from ready queue, sets a timer to interrupt after 1 time quantum & dispatches the process.

Example 1:

Process	BT
P_1	24
P_2	3
P_3	3

time quantum (BT) = 4ms.



WT of $P_1 = 6$

$P_2 = 4$

$P_3 = 7$

Avg. WT = $\frac{17}{3}$ = 5.66ms

TAT of $P_1 = 30$

$P_2 = 7$

$P_3 = 10$

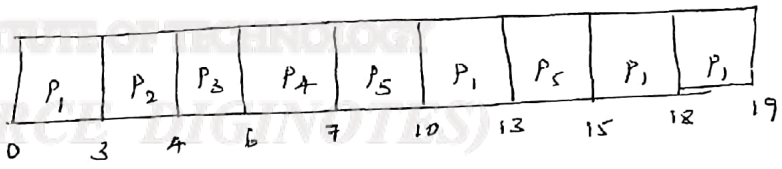
Avg. TAT = $\frac{47}{3}$ = 15.66ms.

Example 2:

Process	BT	AT
P_1	10	0
P_2	1	1
P_3	2	2
P_4	1	3
P_5	5	4

Time slice or TQ = 3ms

Gantt chart:



WT of $P_1 = 15 - 6 = 9$

$P_2 = 3 - 1 = 2$

$P_3 = 4 - 2 = 2$

$P_4 = 6 - 3 = 3$

$P_5 = 13 - 3 - 4 = 6$

Avg. WT = $22/5$ = 4.4ms

TAT of $P_1 = 19$

$P_2 = 4 - 1 = 3$

$P_3 = 6 - 2 = 4$

$P_4 = 7 - 3 = 4$

$P_5 = 15 - 4 = 11$

Avg. TAT = $\frac{41}{5}$ = 8.2ms

Problem:

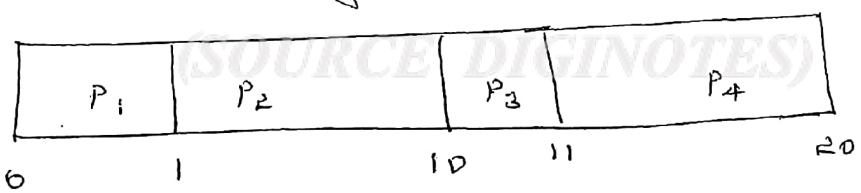
Consider the following set of process that is

Process	AT	BT	Priority
P_1	0	1	1
P_2	1	9	2
P_3	2	1	2
P_4	3	9	3

1. Draw a Gantt charts showing the execution of these processes using FCFS, ~~priority~~ preemptive SJF, non-preemptive SJF, priority & round robin (quantum = 1) scheduling schemes.
2. Compute the turn around time & waiting time for each processes for each of these schemes above.
3. Compute the avg. TA & avg. WT for each algorithm.

Solution:

FCFS Scheduling:



WT of $P_1 = 0$

$P_2 = 1 - 1 = 0$

$P_3 = 10 - 2 = 8$

$P_4 = 11 - 3 = 8$

Avg. WT = $16/4 = 4$ ms

TAT of $P_1 = 1$

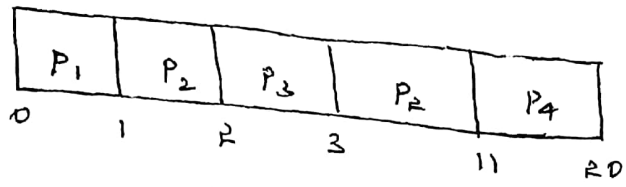
$P_2 = 10 - 1 = 9$

$P_3 = 11 - 2 = 9$

$P_4 = 20 - 3 = 17$

Avg. TAT = $\frac{36}{4} = 9$ ms

Preemptive SJF:



WT of $P_1 = 0$

$P_2 = 3 - 1 - 1 = 1$

$P_3 = 2 - 2 = 0$

$P_4 = 11 - 3 = 8$

Avg. WT $= \frac{9}{4} = 2.25$ ms

TAT of $P_1 = 1$

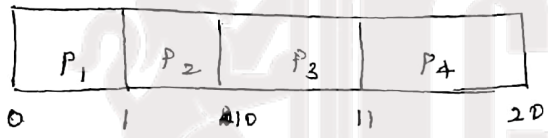
$P_2 = 11 - 1 = 10$

$P_3 = 3 - 2 = 1$

$P_4 = 20 - 3 = 17$

Avg. TAT $= \frac{29}{4} = 7.25$ ms

Non-preemptive SJF:



WT of $P_1 = 0$

$P_2 = 1 - 1 = 0$

$P_3 = 10 - 2 = 8$

$P_4 = 11 - 3 = 8$

Avg. WT $= \frac{16}{4} = 4$ ms

TAT of $P_1 = 1$

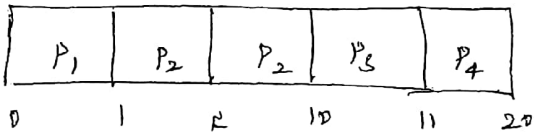
$P_2 = 10 - 1 = 9$

$P_3 = 11 - 2 = 9$

$P_4 = 20 - 3 = 17$

Avg. TAT $= \frac{36}{4} = 9$ ms

Priority Scheduling:



WT of $P_1 = 0$

$P_2 = 1 - 1 = 0$

$P_3 = 10 - 2 = 8$

$P_4 = 11 - 3 = 8$

Avg. WT $= 16/4 = 4$ ms

TAT of $P_1 = 1$

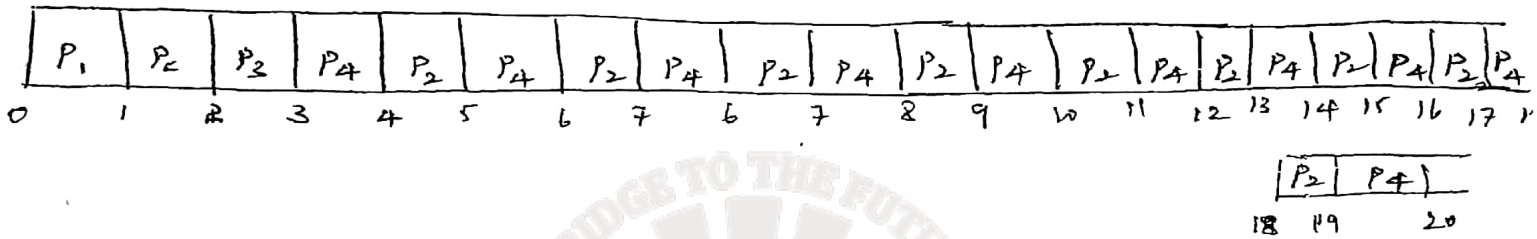
$P_2 = 10 - 1 = 9$

$P_3 = 11 - 2 = 9$

$P_4 = 20 - 3 = 17$

Avg. TAT $= 36/4 = 9$ ms

Round Robin : QT = 1ms.



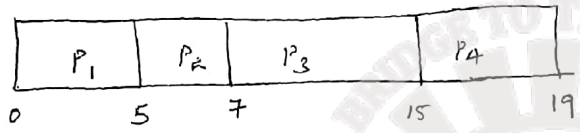
Problem:

Process	AT	BT
P_1	0	5
P_2	0.2	2
P_3	0.6	8
P_4	1.2	4

Find average TAT & WT for the jobs using FCFS, SJF & RR (q=1) algorithms ?

Solution:

FCFS:



WT of $P_1 = 0$

$P_2 = 5 - 0.2 = 4.8$

$P_3 = 7 - 0.6 = 6.4$

$P_4 = 15 - 1.2 = 13.8$

Avg. WT $= \frac{0 + 4.8 + 6.4 + 13.8}{4}$

$= 25/4 = 6.25$ ms

TAT of $P_1 = 5$

$P_2 = 6.8$

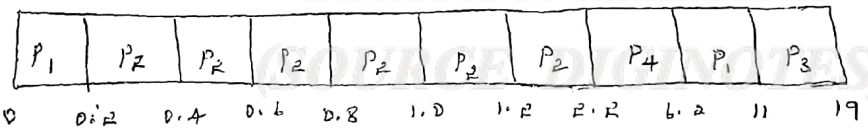
$P_3 = 15 - 0.6 = 14.4$

$P_4 = 19 - 1.2 = 17.8$

Avg. TAT $= \frac{5 + 6.8 + 14.4 + 17.8}{4}$

$= 44/4 = 11$ ms

Preemptive SJF



WT of $P_1 = 6$

$P_2 = 0$

$P_3 = 11 - 0.6 = 10.4$

$P_4 = 2.2 - 1.2 = 1$

Avg. WT $= \frac{6 + 0 + 10.4 + 1}{4}$

$= \frac{17.4}{4} = 4.35$ ms

TAT of $P_1 = 11$

$P_2 = 2.2 - 0.2 = 2$

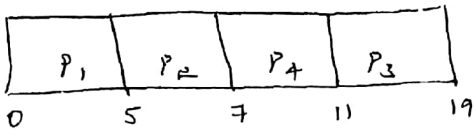
$P_3 = 19 - 0.6 = 18.4$

$P_4 = 6.2 - 1.2 = 5$

Avg. TAT $= \frac{11 + 2 + 18.4 + 5}{4}$

$= \frac{36.4}{4} = 9.1$ ms

Non-preemptive SJF:



WT of $P_1 = 0$

$P_2 = 5 - 0.2 = 4.8$

$P_3 = 11 - 0.6 = 10.4$

$P_4 = 7 - 1.2 = 5.8$

Avg. WT $= \frac{0 + 4.8 + 10.4 + 5.8}{4}$

$= \frac{21}{4} = 5.25$ ms

TAT of $P_1 = 5$

$P_2 = 7 - 0.2 = 6.8$

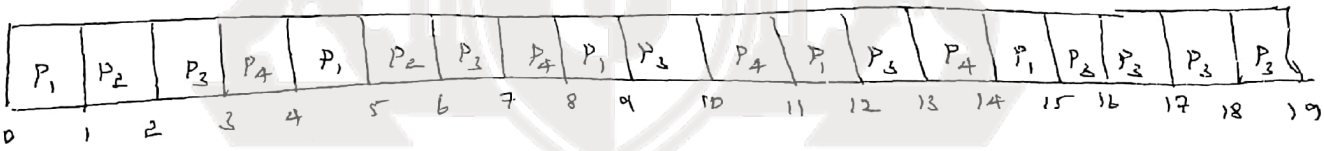
$P_3 = 19 - 0.6 = 18.4$

$P_4 = 11 - 1.2 = 9.8$

Avg. TAT $= \frac{5 + 6.8 + 18.4 + 9.8}{4}$

$= \frac{40}{4} = 10$ ms

Round-Robin (RR): (Given $q = 1$)



WT of $P_1 = 0$

$P_2 = 5 - 1 - 0.2 = 3.8$

$P_3 = 18 - 7 - 0.6 = 10.4$

$P_4 = 13 - 3 - 1.2 = 8.8$

Avg. WT $= \frac{23}{4} = 5.75$ ms

TAT of $P_1 = 5$

$P_2 = 6 - 0.2 = 5.8$

$P_3 = 19 - 0.6 = 18.4$

$P_4 = 14 - 1.2 = 12.8$

Avg. TAT $= \frac{42}{4} = 10.5$ ms