UNIT-2

SYLLABUS:

Process Management: Processes: Process concept, Process Scheduling, Operations on Processes, Inter-process Communication.
Threads: Overview, Multithreading models
CPU Scheduling: Basic Concepts, Scheduling Criteria, Scheduling Algorithms (FCFS, SJF, Priority, RR)

Process Management:

Processes:

Process concept:

Process: Process is the fundamental concept of operating system structure.

- A program under execution is referred as a process.
- Process can also be defined as an active entity that can be assigned to a processor for execution.
- A process is a dynamic object that resides in main memory.
- A process includes the current values of program, counter, processors and registers.
- Each process possess its own virtual CPU.
- A process contains the following two elements.
  1. Program Code
  2. A set of data

Program: Program refers to the collection of instructions given to the system in any programming language.

- A program is a static object residing in a file.
- The spanning time of a program is unlimited.
- A program can exist in a single place in space.
- A program in contrast to a process is a passive entity.
- A program can consists of different types of instructions such as arithmetic instructions, memory instructions and input/output instructions etc.

Process states:

- A program loaded into memory and executing is called process. It is an active entity.
- 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:

State/Activity	Description
New	Process is being created
Running	Instructions are being executed on the processor running
Waiting/blocked	Process is waiting for some event to occur waiting/blocked
Ready	Process is waiting to use the processor ready
terminated	Process has finished execution terminated

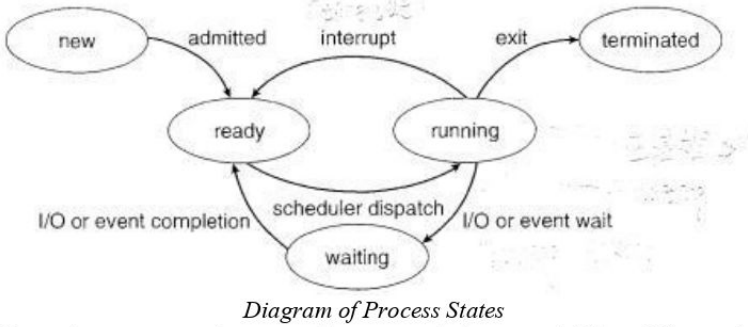


Diagram of Process States

1. **New ->Ready :** OS creates process and prepares the process to be executed,then OS moved the process into ready queue.
2. **Ready->Running :** OS selects one of the Jobs from ready Queue and move them from ready to Running.
3. **Running->Terminated :** When the Execution of a process has Completed, OS terminates that process from running state. Sometimes OS terminates the process for some other reasons including Time exceeded, memory unavailable, access violation, protection Error, I/O failure and soon.
4. **Running->Ready :** When the time slot of the processor expired (or) If the processor received any interrupt signal, the OS shifted Running -> Ready State.
5. **Running -> Waiting :** A process is put into the waiting state, if the process need an event occur (or) an I/O Device require.
6. **Waiting->Ready :** A process in the waiting state is moved to ready state whenthe event for which it has been Completed.

Process Control Block:

- Each process is represented in the operating system by a process control block (PCB).

- It is also be called as task control block.

Process State
Program Counter
CPU Registers
CPU Scheduling Information
Memory – Management Information
Accounting Information
I/O Status Information

Process Control Block

- The components of PCB are
  1) Process state
  2) Program counter
  3) CPU Registers
  4) CPU-Scheduling information
  5) Memory management information
  6) Accounting information
  7) I/O status information

1) **Process state:** The state may be new, ready running, waiting, halted, and so on.
2) **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
3) **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
4) **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
5) **Memory Managemet Information:** This information may include such information as the the page tables, or the segment tables, depending on the system. value of the base and limit registers, memory system used by the operating.
6) **Accounting Information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
7) **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

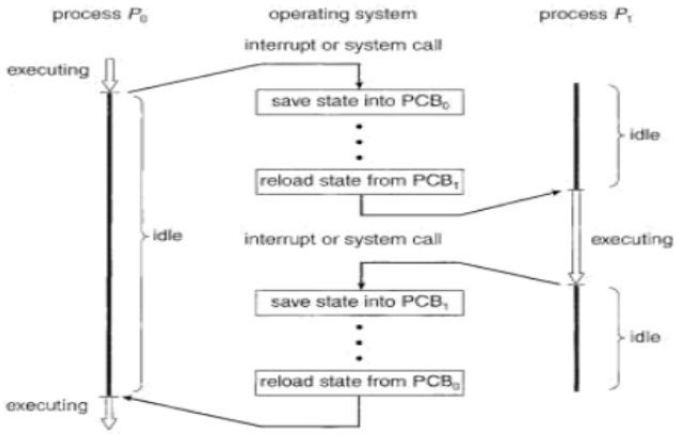
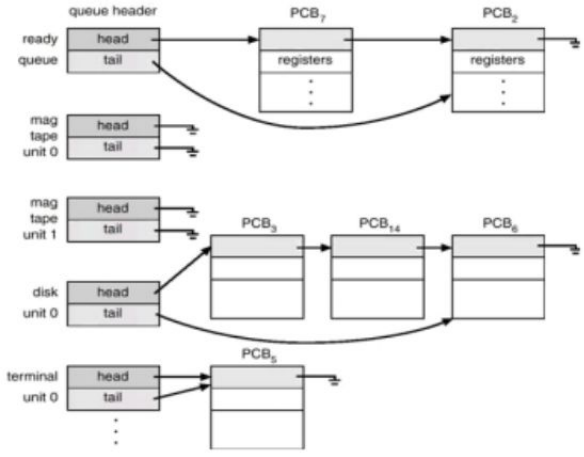


Diagram showing CPU switches from process to process

Process Scheduling:

Q) What is Process Scheduling?

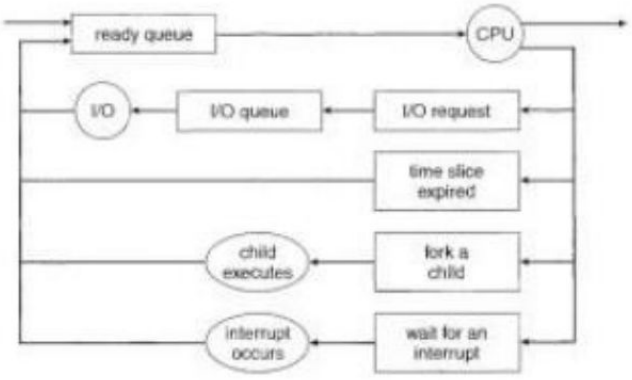
- The objective of CPU is to have some process running at all times, to maximize the CPU utilization.
- In multi-programming systems, a situation often occurs when different programs complete for the processor to be executed first. Since, only one processor is available, two processes cannot be executed simultaneously.
- The objective of time sharing is to switch the CPU among the processors so frequently that users can interact with each program while it is running.
- To meet these objectives, the **process scheduler** selects an available process, possibly from the set of available processes for the program execution on the CPU.
- For a single processor system there will never be more than one running process.
- If there are more than one process, the rest will have to wait until the CPU is free and can be rescheduled.



The ready queue and various I/O device queues

Q) Describe Process Scheduling Queues? Ans:

- There are three types of Queues in which the process is present.
  1) **Job Queue:**As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
  2) **Ready Queue:** The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.
  3) **Device Queue:** The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue.
- A common representation of process scheduling is a queueing diagram,



Queueing Diagram representation of Process Scheduling

- There are two type of queues which are present: The ready queue and set of device queues.
- The circles represent the resources that serve the queues and the arrows indicate the flow of processes in the system.
- 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:
  1) The process could issue an I/0 request and then be placed in an I/O queue.
  2) The process could create a new subprocess and wait for the subprocess's termination.
  3) The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.
- In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Q) Describe Process Schedulers?

(or)

Describe different types of Schedulers?

Ans:

- **Scheduling** is defined as the activity of deciding the resources to be send to the process on their request.
- Scheduler is a program, which selects a user program from disk and allocates CPU to that program.
- A program migrates between the various scheduling queues through out its lifetime.
- The operating system must select, for scheduling purposes, processes from the queues in some fashion.
- The selection of a process from queues is carried out by the appropriate scheduler.
- There are three types of schedulers. They are as follows,
  1. Long Term Schedulers (LTS)
  2. Short Term Schedulers (STS)
  3. Medium Term Schedulers (MTS)

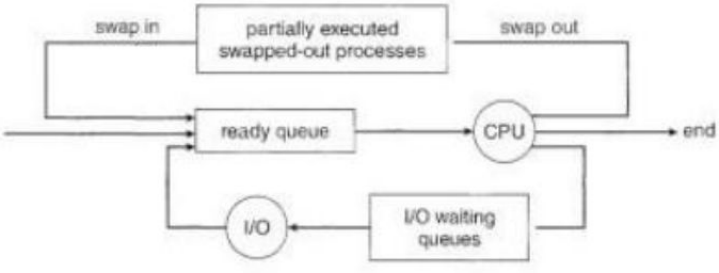
1) Long Term Schedulers (LTS): The long-term scheduler is also called as **job scheduler**, selects processes from this pool and loads them into main memory (ready queue) for execution.

- Long term schedulers is used to decide, which process are to be selected for processing.

- Long term schedulers is defined as program (part of operating system) which selects a job from the disk and transfers it into main memory
- If the number of ready processes in the ready queue becomes very high, the overhead in the operating system for maintaining the long lists, context switching and dispatching over-crosses the limit.
- Therefore, it is useful to let in, only a limited number of processes in the ready queue to complete for the CPU. The long term scheduler manages this.

2) **Short Term Schedulers (STS):** The short-term scheduler also called as **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them. It decides which of the ready processes are to be scheduled or dispatched next. If a process requires an I/O device, which is not present available then process enters device queue. Short term scheduler maintains ready queue and device queue.
   - The difference between Long Term Scheduler (LTS) and Short Term Scheduler (STS) is that the Long Term Scheduler is called less frequently, where as the Short Term Scheduler is called more frequently.
   - Long Term Scheduler must select a program from disk into main memory only once. i.e., when the program is executed.
   - However, Short Term Scheduler must select a job from ready queue often.(for every 1 second, in UNIX operating system) i.e., for every one second the Short Term Scheduler is called, it will select one Process Control Block (PCB) from the ready queue and gives CPU that job.
   - After 1 second is completed, again Short Term Scheduler is called for selecting one more job from the ready queue. This process repeats.
   - Thus, because of the short duration between the executions, the Short Term Schedulers must be very fast in selecting a job, otherwise CPU will sit idle.
   - However, long term schedule is called less frequently so because of duration between executions, Long Term Schedulers can afford to take some time in selecting a good job from disk. A good job is defined as one which is a process mix of CPU bound and I/O bound.
   - An **I/O bound process** is one that spends more of its time during I/O than it sends during its computations.
   - A **CPU bound process** generates I/O requests infrequently, using more of its time during computations.

3) **Medium Term Schedulers (MTS):** The medium term scheduler sometimes removes the partially executed process from memory to reduce the **degree of multi programming**. If process request an I/O device in the middle of the execution, then the process removed from the main memory and loaded into the waiting queue, often to decrease the overhead of CPU and later resumed. When the I/O operation completed, then the job moved from waiting queue to ready queue. These two operations performed by medium term scheduler.
   - When the degree of multi-programming increases, CPU utilization also increases. At one stage, the CPU utilization is maximum for specific number of user programs in memory.
   - At this stage, the CPU utilization drops if the degree of multi-programming is increased.
   - Immediately, operating system observers decrease in CPU utilization and calls Medium Term Scheduler.
   - The Medium Term Scheduler will swap on excess programs from memory and puts on disk. With this the CPU utilization increases.
   - After some time, when some programs leaves memory, Medium Term Schedulers will swap in those programs which were swapped out back into memory and execution stops. This scheme which is known as swapping, is performed by Medium Term Scheduler.
   - The swap in ad swap out must be done at appropriate times by Medium Term Scheduler.



Addition of Medium Term Schedulers to the Queueing Diagram

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

Q) Differentiate between Process Switching and Context Switching?
Ans:

Process Switching: If an executing process is interrupted, the operating system selects another process and changes its state to the running state by taking the control out of an interrupted process and assigning it to the currently executing process.

- The events that can transfer control to the operating system are interrupts and 'trap'.
- Some examples of interrupts include the following.
  i. Clock based interrupts
  ii. Input/Output interrupts
  iii. Memory Faults etc.
- As interrupts are caused due to an event that occurs outside the currently executing process, the control is transferred directly to the interrupt handler, which then shifts it to the operating system routine, where as with the trap, the operating system determines weather the error is fatal or not.
- An example of fatal error is the 'hyperlink' in an application. If we click on the link then it must reach an application, but due to some problem if it does not gt linked to that application, it is said to be of fatal type of error.
- If the error is fatal, then the process being executed is shifted to a terminated state and a process switch occurs.

Context Switching: Context switching refers to the process of switching the CPU to some other process thereby saving the state of the old process and loading the saved state for the new process.

- Context switching is actually an overhead which means that apart from switching no other tasks will be performed.
- Each machine carry different switching speed based on factors such as memory speed, number of registers that are needed to be copied and the presence of special instructions. (For example, a single instruction that can be used to load or store registers).
- The time required to do context switching typically depends on hardware support.
- An example of switch type is a processor that can provide more than one set of registers.
- In context switch, the pointer needs to be changed to the active set of registers.
- The amount of work that is to be done during the process of context switching is more in case of complex operating systems.
- A context switch may occur without changing the state of process being executed.
- Hence, involves the lesser overhead than the situation in which changes in the process states occurs from running to ready or blocked states.
- In case of changes in the process state, the operating system has to make certain changes in its environment which are:
  1) The context associated with the processor along with program counter and other registers are saved.
  2) Updates the Process Control Block (PCB) wit the process being executed. This involves changing the state of the process to one of the available process states. Updating of other fields is also required.
  3) The PCB of this process is moved to some appropriate queue.
  4) Execution of the active process is transferred by selecting of some other process.
  5) Updates the PCB of the chosen process as it includes the changes in its state to running state.
  6) Updates the data structures associated with the memory management which may require the management of the address translation process.
  7) Restores the context of the suspended process by loading the previous values of the PC and other CPU registers.

Thus, the process in which involves a state change, requires considerably more effort than the context switch.

Operation on Process:

process creation:

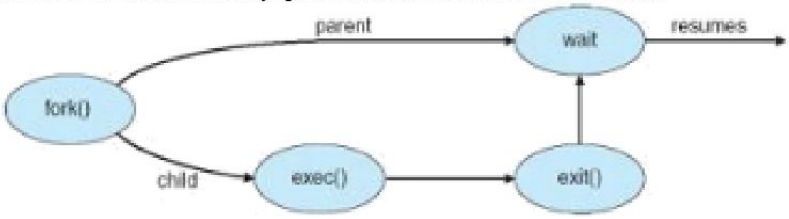
- A process may create several new processes, via a create-process system call, during the course of execution.
- The creating process is called a **parent process**, and the new processes are called the **children of that process**.
- Each of these new processes may in turn create other processes, forming a tree of processes.
- Most operating systems identify processes according to a unique **process identifier (pid)**, which is typically an integer number.
- When a process creates a subprocess, that subprocess can obtain resources from the OS directly or it may be constrained to a subset of the parent process resources. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.
- When a process creates a new process, 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 the same program and data as the parent).
  2. The child process has a new program loaded into it.
- When one process creates a new process, the identity of the newly created process is passed to the parent so that a parent knows the identities of all its children.
- A new process is created by the **fork()** system call.
- Both processes (the parent and the child) continue execution at the instruction after the fork() ,with one difference:
  - The return code for the fork() is zero for the new (child) process and the (nonzero) process identifier of the child is returned to the parent.

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

C Program forking separate process

The **exec()** system call is used after a fork() system call by one of the two processes to replace the process's memory space with a new program.

- The parent can then create more children or if it has nothing else to do while the child runs, it can issue a **wait()** system call to move itself off the ready queue until the termination of the child.



Process Creation

Process Termination:

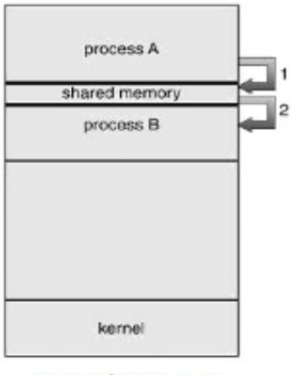
- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the **exit()** system call.
- At that point, the process may return a status value (typically an integer) to its parent process.
- All the resources of the process-including physical and virtual memory, open files, and I/0 buffers-are deallocated by the operating system.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
  1. The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
  2. The task assigned to the child is no longer required.
  3. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. If a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, which is normally initiated by the operating system.

Inter-process Communication:

- Inter Process Communication (IPC) is defined as the communication between process to process.
- IPC provides a mechanism to allow two processes to communicate and to synchronize their actions.
- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- Any process that does not share data with any other process is **independent process**.
- Any process that shares data with other processes is a **cooperating process**.
- There are several reasons for providing an environment that allows process cooperation:
  a) Information sharing
  b) Computation speedup
  c) Modularity
  d) Convenience.
- Cooperating processes require an Inter Process Communication (IPC) mechanism that will allow them to exchange data and information.
- There are two fundamental models of inter process communication:
  1) Shared Memory
  2) Message Passing.

1) Shared Memory Systems:

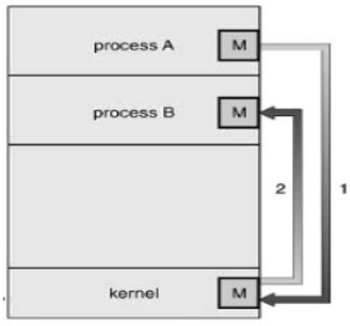
- Inter Process Communication using shared memory model, a region of memory that is shared by cooperating processes is established.
- Processes can then exchange information by reading and writing data to the shared region.
- In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.
- It usually has to deal with problems of mutual exclusion and critical section problem.
- Shared memory allows maximum speed and convenience of communication.
- Shared memory is faster than message passing.
- In shared memory systems, system calls are required only to establish shared-memory regions.
- Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.



Shared Memory

2) Message passing Systems:

- In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.
- Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.
- Message passing is also easier to implement than is shared memory for inter computer communication.
- Message passing is slower than Shared memory.
- Message passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
- A message-passing facility provides at least two operations: send(message) and receive(message).
- Messages sent by a process can be of either fixed or variable size.
- If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them.



Message passing

- Several methods for logically implementing a link and the send0 or receive() operations are listed below:
  1. Direct or indirect communication
  2. Symmetric or asymmetric communication
  3. Automatic or explicit buffering
  4. Send by Copy or Send by Reference
  5. Fixed-sized or Variable-sized messages

❖ **Naming:**

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

1) Direct Communication:

- Each process that wants to communicate must explicitly name the recipient or sender of the communication.
- Here, the send and receive primitives are defined as:
  - **send (P, message)** -Send a message to process P.
  - **receive (Q, message)**-Receive a message from process Q.
- Properties of communication link in this scheme are:
  1. Links are established automatically.
  2. A link is associated with exactly one pair of communicating processes.
  3. Between each pair there exists exactly one link.
  4. The link may be unidirectional, but is usually bi-directional.
- The above scheme exhibits symmetry as both the sender and receiver must name the other for communication.
- A slight variation of the above scheme exhibits asymmetry in addressing. Here, only the sender names the recipient but, the recipient need not name the sender. The send and receive primitives for this scheme are:
  - **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.
- The disadvantage in both symmetric and asymmetric schemes is the limited modularity of the resulting process definitions.

2) Indirect Communication:

- With indirect communication, the messages are sent to and received from mailboxes, or ports.
- A mailbox can be viewed abstractly as an object which messages can be placed by processes and 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 number of different mailboxes. Two processes can communicate only if they share a mailbox. The send and receive primitives are defined as follows:
  - The messages are sent to and received from mailboxes, or ports.
  - **send (A, message)** -Send a message to mailbox A.
  - **receive (A, message)**-Receive a message from mailbox A.
- A communication link in this scheme has the following properties:
  1. Link established only if processes share a common mailbox.
  2. A link may be associated with many processes.
  3. Each pair of processes may share several communication links, each communication link corresponding to one mailbox..
  4. Link may be unidirectional or bi-directional.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox). Since each mailbox

has a unique owner, there can be no confusion about who should receive a message sent to this mailbox. 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 the mailbox no longer exists. Also, a mailbox owned by the operating system is independent and is not attached to any particular process and must provide a mechanism that allows a process to

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

❖ **Synchronization:**

- Communication between processes takes place by calls to send and receive primitives.
- Message passing may be either blocking or non blocking also known as synchronous and asynchronous.
- **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send:** The sending process sends the message and resumes operation.
- **Blocking receive**: The receiver blocks until a message is available.
- **Nonblocking receive:** The receiver retrieves either a valid message or a null.

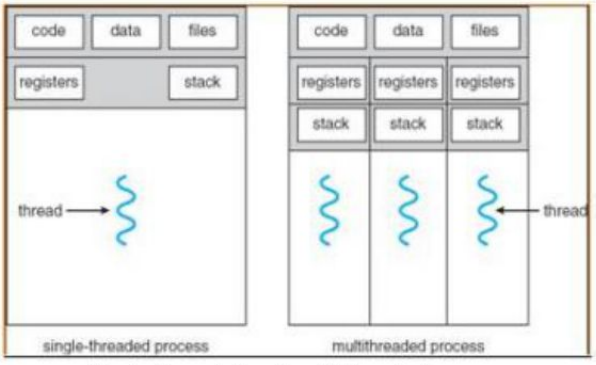
❖ **Buffering:**

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
- Basically, such queues can be implemented in three ways:
  1. Zero capacity
  2. Bounded capacity
  3. Unbounded capacity
     1) **Zero capacity:** The queue has a maximum length of zero.
     2) **Bounded capacity:** The queue has finite length n.
     3) **Unbounded capacity:** The queue's length is potentially infinite.
- The zero-capacity is sometimes is referred as a message with no buffering; other cases are referred as the system with automatic buffering.

Threads:

Overview

- A thread (also called as lightweight process) is a basic unit of CPU utilization;
- It consists of program counter, register set and stack space.
- Multiple threads can be created by a particular process.
- A thread shares code section, data section and OS resources (open files, signals) with peer threads.
- However, each of them will have their separate register set values and stack.
- Consider an example of pagemaker or any word processing application. It has two important threads executing in it, one to read the keystrokes from the keyboard and other a spelling and grammar checking thread running in background.
- The following diagram shows process having single thread and multithreaded process.



Single threaded and multithreaded process

❖ **Thread States:**

1) **Born State :** A thread is just created.
2) **Ready State :** The thread is waiting for CPU.
3) **Running :** System assigns the processor to the thread.
4) **Sleep :** A sleeping thread becomes ready after the designated sleep time expires.
5) **Dead :** The Execution of the thread finished.

❖ **Benefits of Threads:**

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

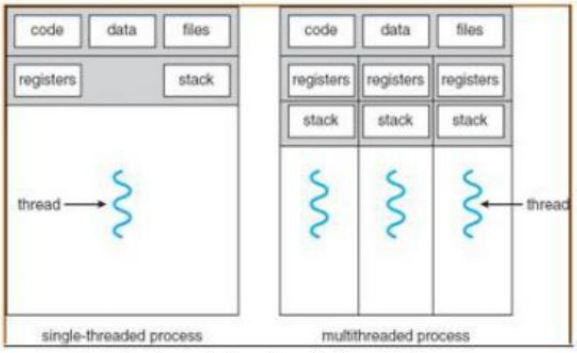
❖ **Types of Threads:**

1. Kernel-supported threads (Mach and OS/2)
2. User-level threads
3. Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).

Multithreading models

- A process is divided into number of smaller tasks each task is called a **Thread**.
- Number of Threads with in a Process execute at a time is called **Multithreading**.

- If a program, is multithreaded, even when some portion of it is blocked, the whole program is not blocked. The rest of the program continues working if multiple CPU's are available.
- Multithreading gives best performance. If we have only a single thread, number of CPU's available, No performance benefits achieved.
- A thread is a basic unit of CPU utilization.
- A traditional process has a single thread of control.
- If a process has multiple threads of control, it can perform more than one task at a time.



Single threaded and multithreaded process

- The **benefits of multithreaded programming** are
  1. Responsiveness
  2. Resource sharing
  3. Economy
  4. Scalability.
- Process creation is heavy-weight, while thread creation is light-weight.
- Can simplify code, increase efficiency.

1) Responsiveness:

- Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

2) Resource sharing:

- Processes may only share resources through techniques such as shared memory or message passing.
- Threads share the memory and the resources of the process to which they belong by default.
- The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3) Economy:

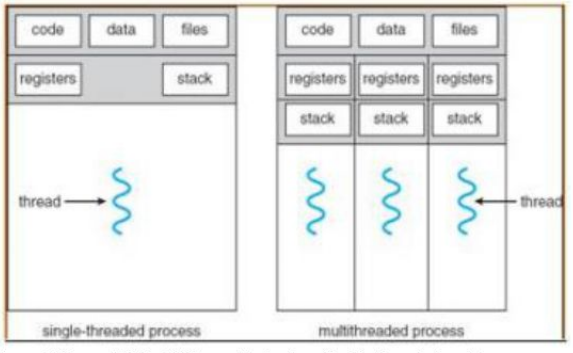
- Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads

4) Scalability:

- The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors.
- A single-threaded process can only run on one processor, regardless how many are available.
- Multithreading on a multi CPU machine increases parallelism.

Multithreading models:

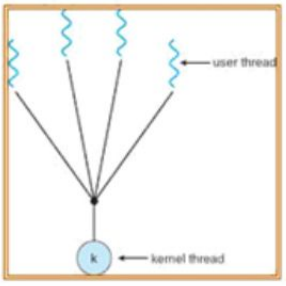
- A thread is a basic unit of CPU utilization.
- A traditional process has a single thread of control.
- If a process has multiple threads of control, it can perform more than one task at a time.
- The benefits of multithreaded programming are Responsiveness, Resource sharing, Economy, scalability.
- 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 and are managed without kernel support.
- Kernel threads are supported and managed directly by the operating system.
- A relationship must exist between user threads and kernel threads.



- Different multi threaded models exhibit different kinds of relationships the user and kernel threads may have. They are:
  1. Many-to-One Model
  2. One-to-One Model
  3. Many-to-Many Model

1) Many-to-One Model:

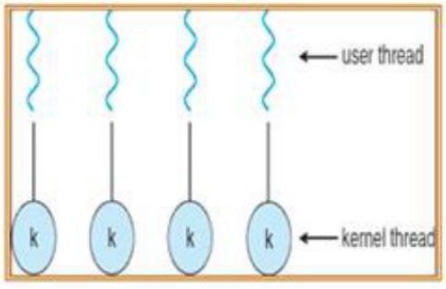
- The many-to-one model maps many user-level threads to one kernel thread.
- Thread management is done by the thread library in user space, so it is efficient.
- But the entire process will block if a thread makes a blocking system call.
- Multiple threads are unable to run in parallel on multiprocessors.
- True concurrency is not gained because the kernel can schedule only one thread at a time.



Many-to-One Model

2) <u>One-to-One Model:</u>

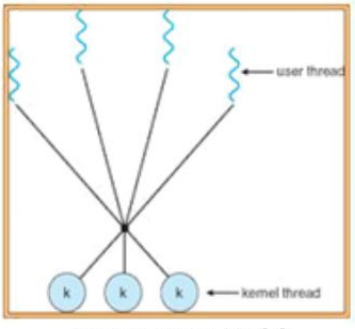
- The one-to-one model maps each user thread to a kernel thread.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application.
- It allows for greater concurrency, but the developer has to be careful not to create too many threads within an application



One-to-One Model

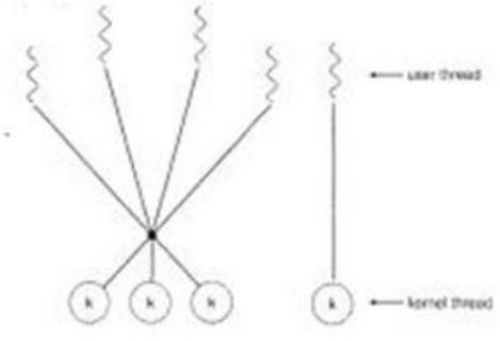
3) <u>Many-to-Many Model:</u>

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine.
- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Many-to-Many Model

- **<u>Two-level model:</u>** multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation sometimes referred to as the two-level model.

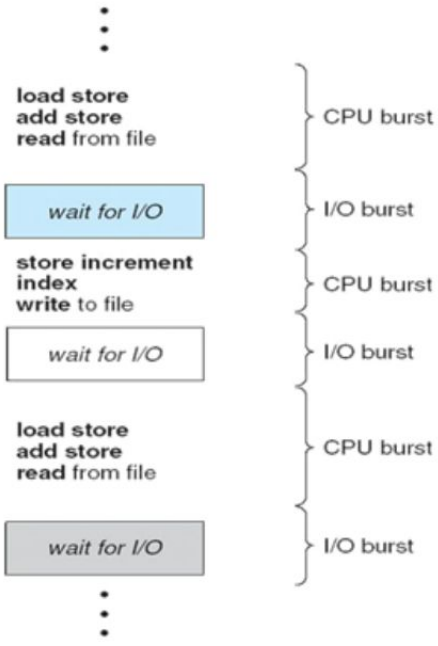


Two-level Model

CPU Scheduling:

Basic Concepts:

- **CPU-I/O Burst Cycle:** Scheduling is a fundamental operating-system function. Process execution consists of a cycle of CPU execution and I/O wait where, processes alternate between a CPU burst and an I/O burst.



Alternating Sequence of CPU and I/O bursts

- **CPU Scheduler:** A CPU scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them. CPU scheduling decisions may take place when a process:
  1) Switches from running state to waiting state {I/O request or waiting for a child process}
  2) Switches from running state to ready state {when an interrupt occurs}
  3) Switches from waiting state to ready {completion of I/O}
  4) Terminates
  - Scheduling taken place under circumstances 1 and 4 is called as **nonpreemptive** or **cooperative** scheduling and otherwise its called as **preemptive**.
  - In **nonpreemptive scheduling**, once the CPU has been allocated to a process, the CPU is kept by it, until it releases the CPU either by terminating or by switching to the waiting state.
  - In **preemptive scheduling**, process may have CPU taken away before completion of current CPU burst.
- **Dispatcher** is the module that gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program.
- **Dispatch latency** is the time taken by the dispatcher to stop one process and start another running.

Scheduling Criteria:

- Different CPU-scheduling algorithms have different properties and may favor one class of processes over another.
- Different criteria are taken into account for comparison of CPU-scheduling algorithms. They are:
  - **CPU utilization** – keep the CPU as busy as possible
  - **Throughput** – One measure of work is the number of processes completed per time unit, called throughput.
  - **Turnaround time** – The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
  - **Waiting time** – Waiting time is the sum of the periods spent waiting in the ready queue.
  - **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
- ➢ The criteria for optimization of a CPU scheduling algorithm are Max CPU utilization, Max throughput, Min turnaround time, Min waiting time, Min response time.

Scheduling Algorithms:

CPU scheduling algorithms

First-Come, First-Served (FCFS) Scheduling:

- The simplest and a **non-preemptive CPU scheduling algorithm** is the **first-come, first-served (FCFS) scheduling algorithm.**
- Here, the process that requests the CPU first is allocated the CPU first.
- The implementation is easily managed by a FIFO queue.
- When the process enters the ready queue, its PCB is linked to the tail of the queue.
- When CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
- The average waiting time under FCFS policy, is often quite long.
- Take an example of three process with their necessary CPU-burst time given in milliseconds.

Example for FCFS Scheduling:

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds for the processes P_1, P_2, P_3:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the process arrive in the order P_1 , P_2 , P_3, and are served as FCFS order, then the resultant Gantt Chart, which is a bar chart that illustrates a particular schedule the start and finish times of each of the processes.

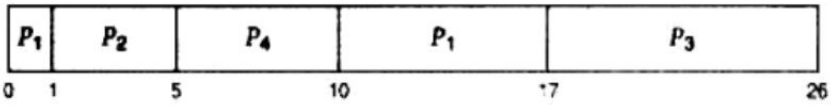
So, The waiting time for process *P1* = 0 milliseconds
The waiting time for process *P2* = 24 milliseconds
The waiting time for process *P3* = 27 milliseconds .
Thus, the average waiting time = (0+ 24 + 27)/3 = 17 milliseconds.
But, if the processes arrive in the order P_2, P_3, P_1, then we get the following Gantt chart,

Here, The waiting time for process *P1* = 6 milliseconds
The waiting time for process *P2* = *0* milliseconds
The waiting time for process *P3* = *3* milliseconds .
Now, the average waiting time is (6 + 0 + 3)/3 = 3 milliseconds, which is a substantial reduction.
So, the average varies with the variation in process CPU burst time.

- Another difficulty with FCFS is, it tends to favour CPU bound process with I/O bound process. Consider that there is a collection of processes one of which mostly uses CPU and a number of processes which uses I/O devices. When a CPU bound process is running all the I/O bound processes must wait, which causes the I/O devices to be idle. After finishing its CPU operation, the CPU bound process moves to an I/O device. Now, all the I/O bound processes, very short CPU bursts execute quickly and move back to I/O queues and causes the CPU to sit idle. In this way FCFS may result in different use of both processor and I/O devices.
- Once the CPU has been allocated to a process, it will not release the CPU until it is terminated or switched to the waiting state. So, this algorithm is non-primitive. It is difficult to implement for time sharing systems in which each user gets the CPU on a time based sharing.
- If there is one CPU bound process and many I/O bound processes, and if the CPU bound process gets hold of the CPU, the other processes have to wait for the big process to get off the CPU. This is called a **convoy effect**.
- This convoy effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Shortest-Job-First (SJF) scheduling algorithm:

- In this approach, we associate with each process the length of its next CPU burst.
- Use these lengths to schedule the process with the shortest time.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. This is also called as "shortest next CPU burst".



Process P1 is started at time 0, since it is the only process in the queue.
Process P2 arrives at time 1.
The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.
The average waiting time for this example is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5 milliseconds.
A non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

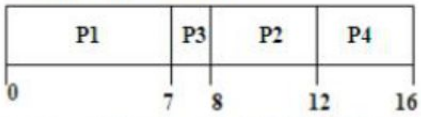
Example for Shortest Job First (SJF) and Shortest-Remaining-Time-First (SRTF) Scheduling:

Consider the following arrival time and burst time for processes P1, P2, P3,P4:

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-Preemptive SJF Scheduling

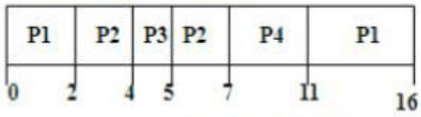
Gantt Chart for Schedule



Average waiting time is (0+6+3+7)/4 = 4

Preemptive SJF Scheduling

Gantt Chart for Schedule



Average waiting time is (9+1+0+2)/4 = 3

If all the jobs are of the same length, then SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length).

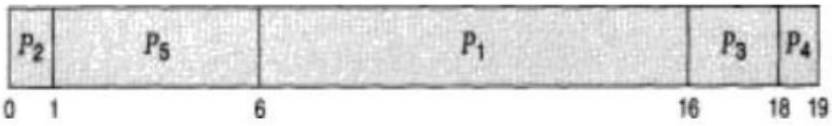
Priority Scheduling Algorithm:

- The algorithm associates each process with a priority, the process with highest priority will get the CPU first.
- If the re are two processes with the same priority, FCFS scheduling is used to break the tie.
- An SJF algorithm is simply a priority algorithm, where the priority (p) is the inverse of the next (predicted) CPU burst. The larger the CPU burst, the lower the priority and vice versa.

Example for Priority Scheduling:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



So, The waiting time for process *P1* = 6 milliseconds
The waiting time for process *P2* = *0* milliseconds
The waiting time for process *P3* = *16* milliseconds
The waiting time for process *P4* = *18* milliseconds.
The waiting time for process *P5* = *1* milliseconds.
Thus, the average waiting time = (6 + 0 + 16 + 18 + 1)/ 5 = 41/5 = 8.2 milliseconds.

- The allotment of priorities can be carried out internally as well as externally.
- In an internally defined priorities are alloted based on the computation of certain measurable quantities such as CPU burst time, time limits etc.
- In the externally defined priority, these are based on certain external factor associated with the process.
- An example of certain priorities based on the importance of the process.
- Similar to the SJF, priority scheduling can also be used to preemitive and non-preemitive. But, major drawback associated with this algorithm is indefinite blocking which is also called as **starvation**.
- In this case the process with lowest priority will never get the CPU because it keeps on alloting the highest priority to other processes. To avoid this, a technique called "**Aging**" is employed which increases the priority of processes that are present in the ready queue for a long time.

Round Robin scheduling algorithm:

- The round-robin (RR) scheduling is similar to FCFS but with preemption added to switch the processes.
- The round-robin (RR) scheduling algorithm is designed especially for time sharing systems.
- Each process gets a small unit of the CPU time called the **time slice** or **time quantum**, which is usually 10-100 milliseconds.
- The ready queue is treated as a circular queue.

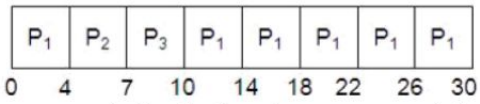
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- If a process CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The round robin (RR) scheduling algorithm is **preemptive scheduling algorithm.**
- After this time has elapsed and if the process hasn't finished execution, the timer sets an interrupt to the operating system.
- A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.
- It is a preemptive scheduling algorithm.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n\text{-}1)q$ time units.

Example for Round Robin Scheduling:

Consider the following example, where the time quantum is 4 milliseconds.

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:



Process P_1 gets the first 4 milliseconds and then its preempted after the first time quantum and the CPU is given to the next process.

Since process P_2 does not need 4 milliseconds, it quits before its time quantum expires.

The CPU is then given to the next process, process P_3.

Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum.

So, The waiting time for process *P1 = (10-4)* = 6 milliseconds

The waiting time for process *P2 = 4* milliseconds

The waiting time for process *P3 = 7* milliseconds

Thus, the average waiting time = (6 + 4 + 7)/3 = 17/3 = 5.66 milliseconds.