UNIT 2/ Module 2: Process Management

Chapter-3: Process Concept

Contents:

3.1 Process concept

3.2 Process scheduling;

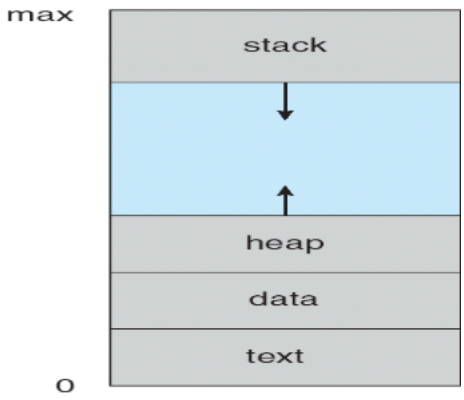
3.1 The Process concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Program is a passive entity.
- A process is a program in execution.
- An active entity that can be assigned to and executed on a processor.
- A process is unit of work.
- It has limited time span.
- A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task.

3.1.1The Process

- A program is a passive entity i.e. a file containing a list of instructions stored on disk called as executable file.
- A process is an active entity with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes process when an executable file is loaded to memory .Two common techniques for loading executable files are:
  - Double click on icon representing executable file
  - Entering the name of the executable file on the command line(ex prog.exe or a.out)
- A process includes:

- Text section (Program code)
- Value of program counter and contents of processor's registers (PCB)
- Stack (temporary data like function parameters, return addresses, local variables)
- Data section (global variables)
- Heap (memory dynamically allocated to process to hold intermediate computation data at run time)



Process in memory

- Two processes may be associated with same program but they are not considered as separate processes e.g. many copies of one mail program.
- A process may spawn many processes as it runs

3.1.2 Process State

- As the 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.
  - **ready**: The process is waiting to be assigned to a processor.
  - **running**: Instructions are being executed.
  - **waiting**: The process is waiting for some event to occur (I/O or reception of signal)
  - **terminated**: The process has finished execution.
- Only one process can be running on any processor at any instant.

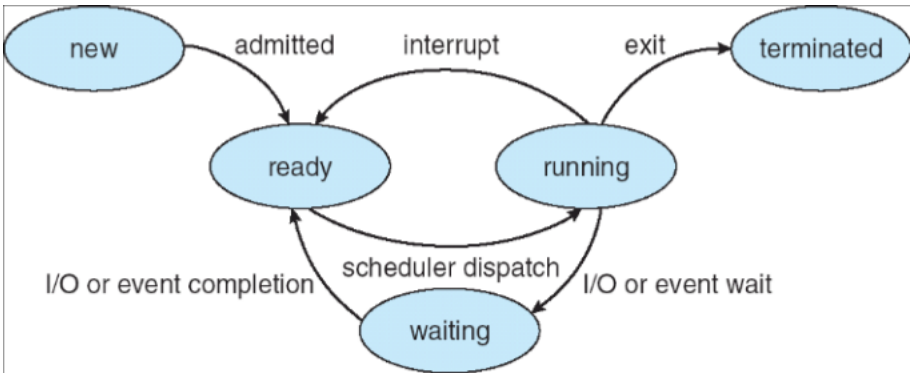
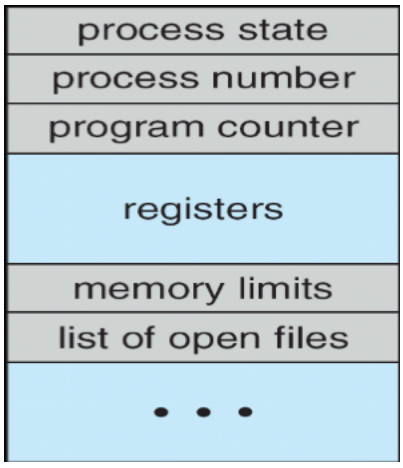


Diagram of process state

3.1.3 Process Control Block (PCB)

- Each process is represented by a process control block (PCB) also called as ***task control block.***



Process Control Block(PCB)

- PCB is repository of information
- **Information associated with each process:**
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information

- ❖ Accounting information
- ❖ 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 next instruction to be executed for this process.
3. **CPU registers**
    - ➢ Number and type of registers
    - ➢ They include accumulators, index registers, stack pointers, general purpose registers and 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 includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
5. **Memory-management information**
    - ➢ Value of base and limit registers, page table or segment table
6. **Accounting information**
    - ➢ Amount of CPU and real time used, time limits, account numbers , job or process numbers.
7. **I/O status information**
    - ➢ List of I/O devices allocated to the process
    - ➢ list of open files

3.2 Process Scheduling:

- ❖ The objective of multiprogramming is to have some processes running at all time, to

maximize CPU utilization.

- The objective of time sharing is to switch CPU among the processes so frequently that users can interact with each process while it is running.
- To meet these objectives, *process scheduler* selects an available process for execution on CPU.
- For a single-processor system, there will never be more than one running process,.
- If there are more processes, the rest will have to wait until the CPU is free and can be reshecduled.

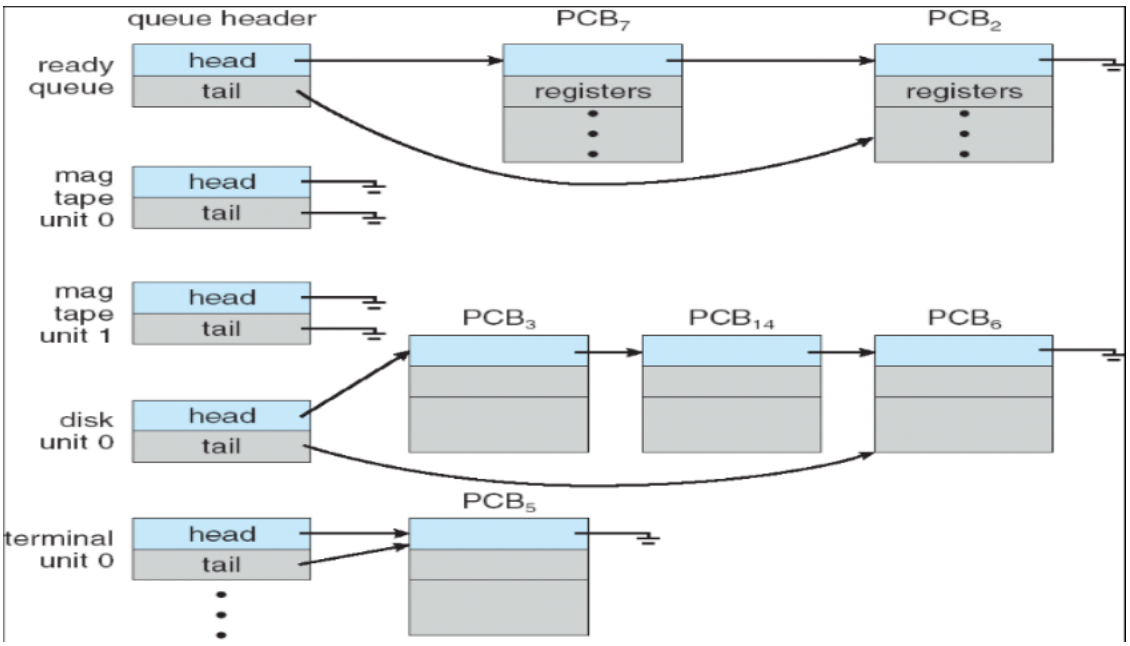


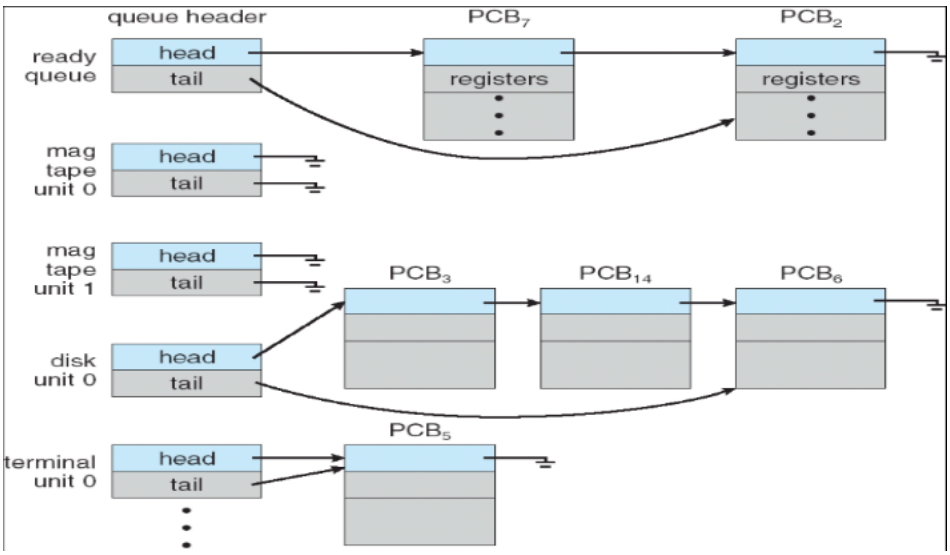
Fig: The ready queue and various I/O device queues

3.2.1 Scheduling Queues:

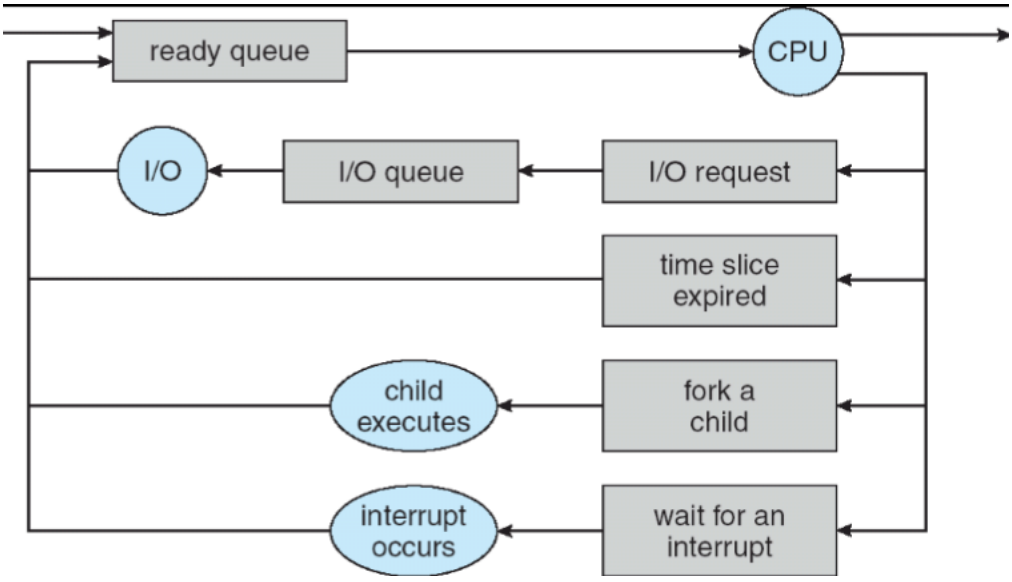
- **Job queue**
  - Set of all processes in the system
- **Ready queues**
  - Set of all processes residing in main m
  - emory, ready and waiting to execute
- **Device queues**

- Set of processes waiting for an I/O device

- Processes migrate among the various queues
- Ready queue
- Set of all processes residing in main memory, ready and waiting to execute.
- Header of the queue contains the pointers to first and last PCB.
- Each PCB contains a pointer to next PCB.
- I/O Device queues
- Set of processes waiting for an I/O device



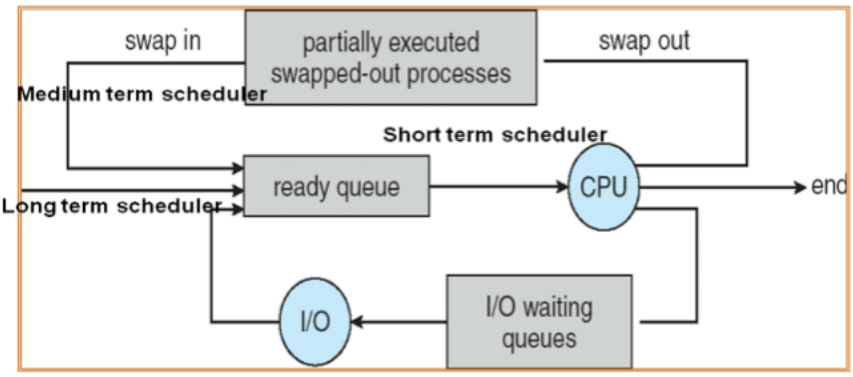
- A new process initially put in ready queue, where it waits for CPU for execution.
- Once process is allocated CPU, following events may occur:
  - The process issues an I/O request, and then be placed in an I/O device queue.
  - The process creates a new subprocess and waits for its termination.
  - The process is removed forcibly from the CPU as a result of interrupt or expired time slice.
  - The process ends.



- A process migrates between the various scheduling queues throughout its life time.
- The operating system must select the processes from or to the scheduling queues.
- The selection process is carried out by schedulers.

Process Scheduling: Types of Schedulers

- **Long-term scheduler (or job scheduler)**
  - Selects which processes should be brought into the ready queue.
- **Short-term scheduler (or CPU scheduler)**
  - Selects process from ready queue which should be executed next and allocates CPU.
- **Medium-term scheduler**
  - Swap in the process from secondary storage into the ready queue.



Process Scheduling: Schedulers

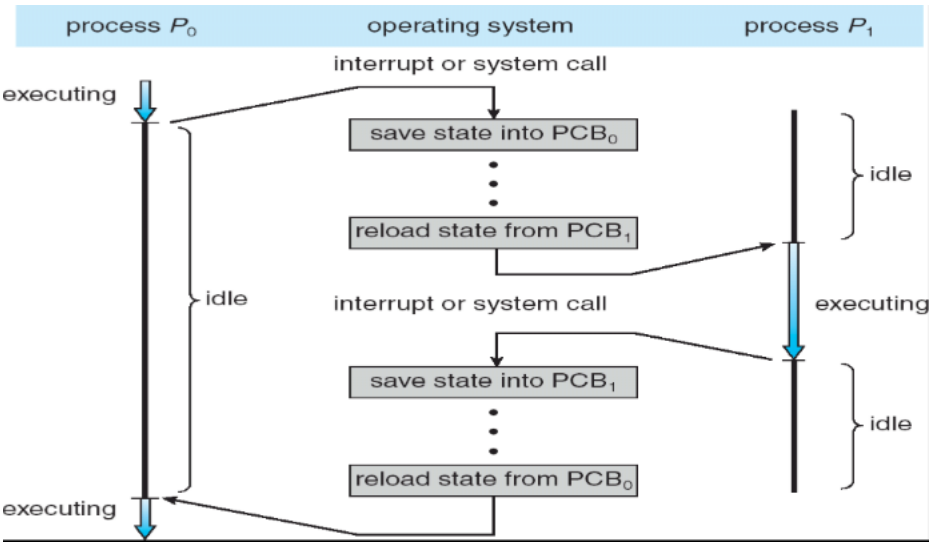
- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast).
- Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow).
- The long-term scheduler controls the degree of multiprogramming.
- Processes can be described as
    - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts may be needed.
    - CPU-bound process – spends more time doing computations; long CPU bursts are required.
- System should have mix of both type of processes so I/O waiting queue and ready queue will have equal and balanced work load.

Context Switch

- Many users processes and system processes run simultaneously.
- Switching CPU from one process to another process is context switch.
- The system must save the state of the old process and load the saved state of the new process.
- Context-switch time is overhead and the system does no useful work while switching.
- Switching speed varies from machine to machine depending upon memory, number of registers and hardware support.
- Information in saved process image
    - User data

- Program counter
- System stack
- PCB

CPU Switch from Process to Process



Questions

- **What is a process? Explain the different states with a process state diagram.**
- **Explain Process Control Block.**
- **What is a scheduler? Explain different types of schedulers.**
- **What is context switching? Explain process context switching with diagram.**
- **Discuss the information and data needed for the process.**
- **What is scheduling queue? Explain with example.**

UNIT 2/ Module 2:Process Management
Chapter-4 : Multithreaded Programming

Contents:

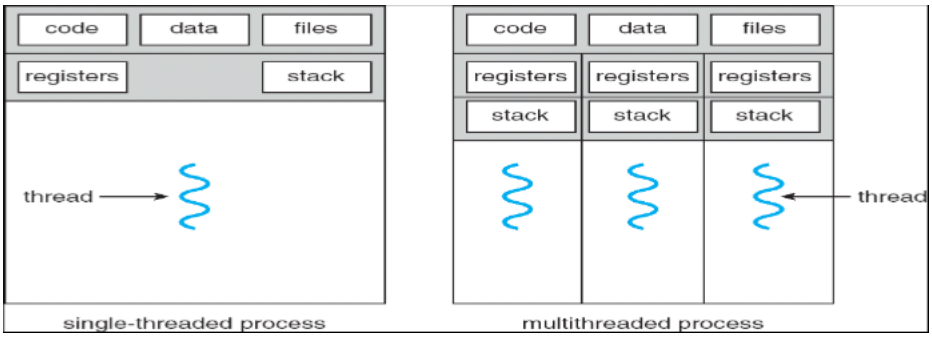
- Overview
- Multithreading Models

4.1 : Overview

Multithreaded Processes

- If one application should be implemented as a set of related units of execution. Two possible ways:
- Multiple processes
    - Heavy weight
    - Less efficient
- Multiple threads
- A thread
    - is referred as a light weight process (LWP).
    - is a basic unit of CPU utilization
    - has a thread ID, a program counter, a register set, and a stack.
    - shares a code section, data section, open files etc. with other threads belonging to the same process.

Single and Multithreaded Processes



4.1.1 Multithreaded Processes

- A traditional (or heavy weight) process has a single thread of control.
- If a process can have multiple threads, it can do more than one task.
- Multiple threads
  - Light weight because of shared memory
  - Thread creation and termination is less time consuming
  - Context switch between two threads takes less time
  - Communication between threads is fast because they share address space, intervention of kernel is not required.
- Multithreading refers to the ability of an OS to support multiple threads of execution within a single process.
- A multithreaded process contains several different flows of control within the same address space.
- Threads are tightly coupled.
- A single PCB & user space are associated with the process and all the threads belong to this process.
- Each thread has its own thread control block (TCB) having following information:
  - ID, counter, register set and other information.

4.1.2 Why multithreading

- Use of traditional processes incurs high overhead due to process switching.

- Two reasons for high process switching overhead:
  - Unavoidable overhead of saving the state of running process and loading the state of new process.
  - Process is considered to be a unit of resource allocation, resource accounting and interprocess communication.
- Use of threads splits the process state into two parts – resource state remain with the process while execution state is associated with a thread.
- The thread state consists only of the state of the computation.
  - This state needs to be saved and restored while switching between threads of a process. The resource state remains with the process.
  - Resource state is saved only when the kernel needs to perform switching between threads of different processes.

Examples

- Web browser
  - Displaying message or text or image
  - Retrieves data from network
- Word processor
  - Displaying graphics
  - Reading key strokes
  - Spelling and grammar checking

4.1.3 Benefits of Multithreading

- Responsiveness
  - A process continues running even if one thread is blocked or performing lengthy operations, thus increased response.
- Resource Sharing
  - Threads share memory and other resources.
  - It allows an application to have several different threads of activity, all

within the same address space.

- Economy
  - Sharing the memory and other resources
  - Thread management (creation, termination, switching between threads) is less time consuming than process management.
- Utilization of MP Architectures
  - Each thread may be running in parallel on different processor – increases concurrency.
  - A single threaded process can run on only one processor irrespective of number of processors present.

4.2 Multithreading Models: User & Kernel Threads

- Support for threads may be provided either at user level i.e. user threads or by kernel i.e. kernel threads.
- User threads are supported above the kernel and are managed without kernel support.
- Kernel threads are supported by OS.

Multithreading Models: User Threads

- These are supported above the kernel and are implemented by a thread library at user level.
- Library provides thread creation, scheduling management with no support from kernel.
- No kernel intervention is required.
- User level threads are fast to create and manage.
- Examples:
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*

Multithreading Models: Kernel Threads

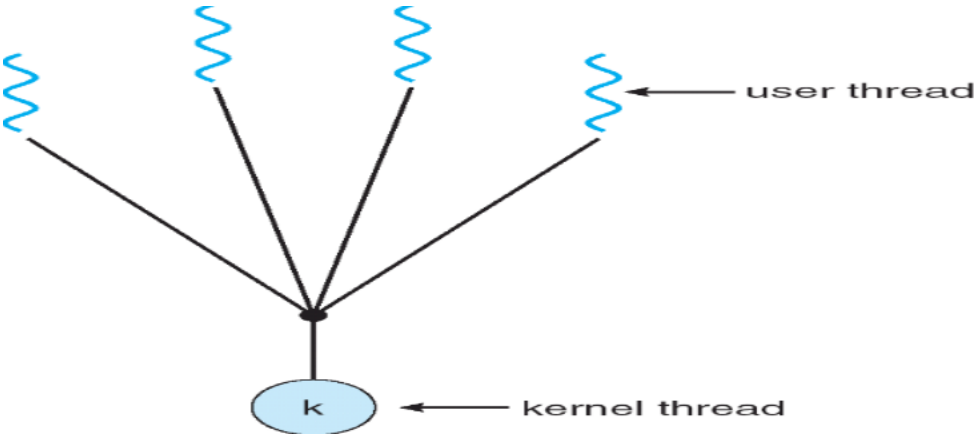
- Kernel performs thread creation, scheduling, management in kernel space.
- Kernel threads are slower to create and manage.
- If a thread performs a blocking system call, the kernel can schedule another kernel thread.
- In multiprocessor environment, the kernel can schedule threads on different processors.
- Most of the operating system support kernel threads also:
  - Windows XP
  - Solaris
  - Mac OS X
  - Tru64 UNIX
  - Linux

Multithreading Models

- There must exist a relationship between user threads and kernel threads
  - Many-to-One
  - One-to-One
  - Many-to-Many

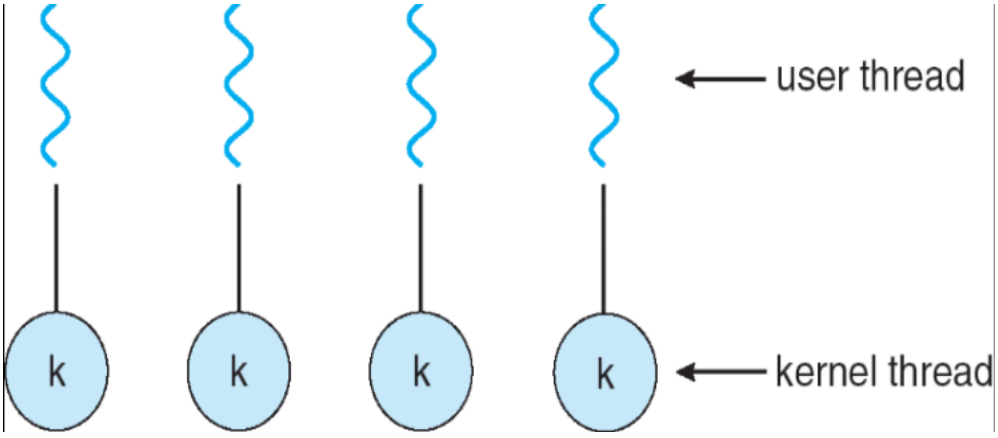
4.2.1 Many-to-One Model

- Many user-level threads are mapped to single kernel thread.
- Thread management is done by the thread library in user space.
- It is fast, efficient.
- Drawback: If one user thread makes a system call and only one kernel thread is available to handle, entire process will be blocked.
  - Green threads: a library for Solaris
  - GNU Portable Threads



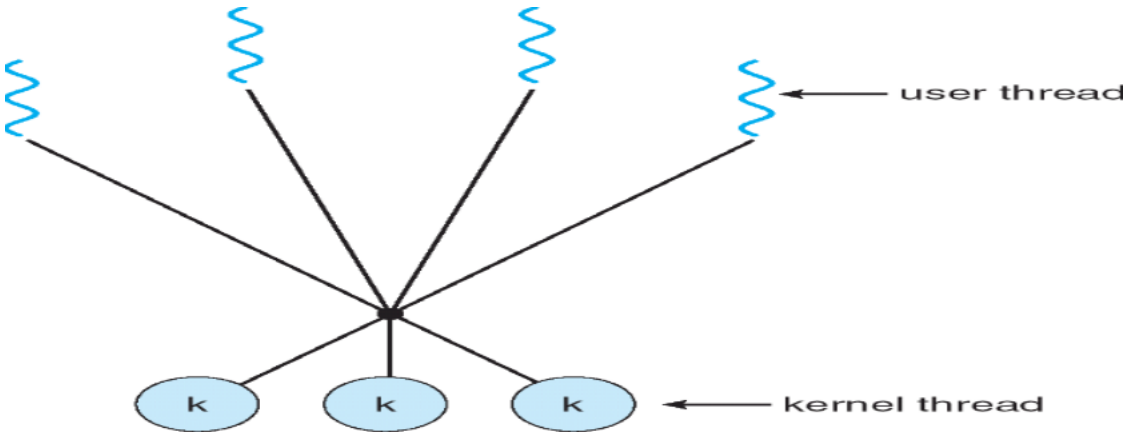
4.2.2 One-to-One Model

- Each user-level thread is mapped to one kernel thread
- It provides more concurrency than many to one model.
- If blocking call from one user thread to one kernel thread, another kernel thread can be mapped to other user thread.
- It allows multiple threads to run in parallel.
- Drawback: Creating a user thread requires the creation of corresponding kernel thread which is overhead.
    - Windows 95/98/NT/2000/XP
    - Linux
    - Solaris 9 and later



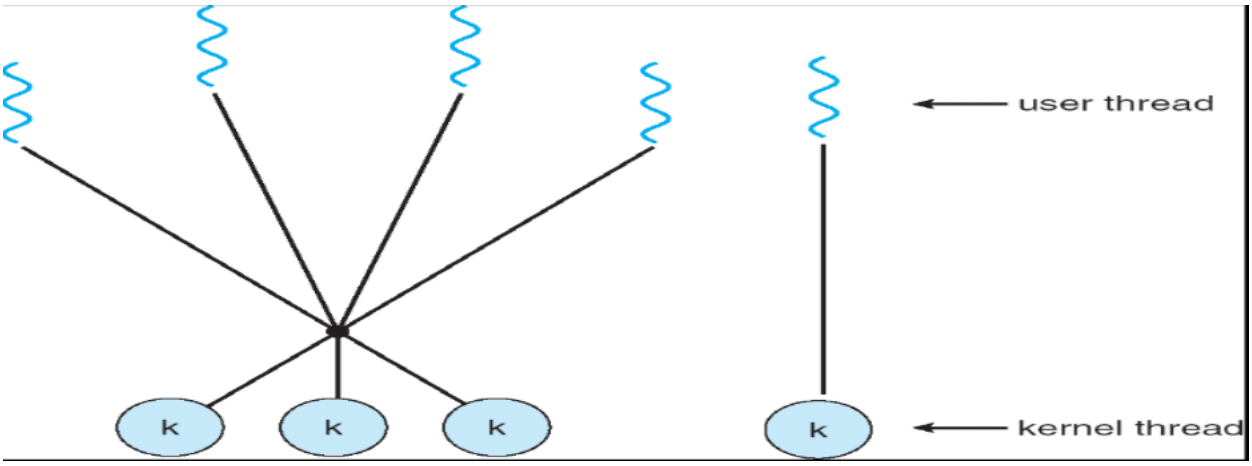
4.2.3 Many-to-Many Model

- Many user-level threads are mapped to a smaller or equal number of kernel threads.
- Number of kernel threads may be specific to either application or a particular machine.
- It allows many user level threads to be created.
- Corresponding kernel threads can run in parallel on multiprocessor.



Variation of Many-to-Many Model: two-level model

- Two-level model is a variation of Many-to-Many which has one-to-one also.
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Questions

- What is thread? Explain the concept of multithreading
- Explain various models of multithreading.
- Why multithreading is useful?
- Discuss benefits of multithreading.
- Describe user and kernel thread.

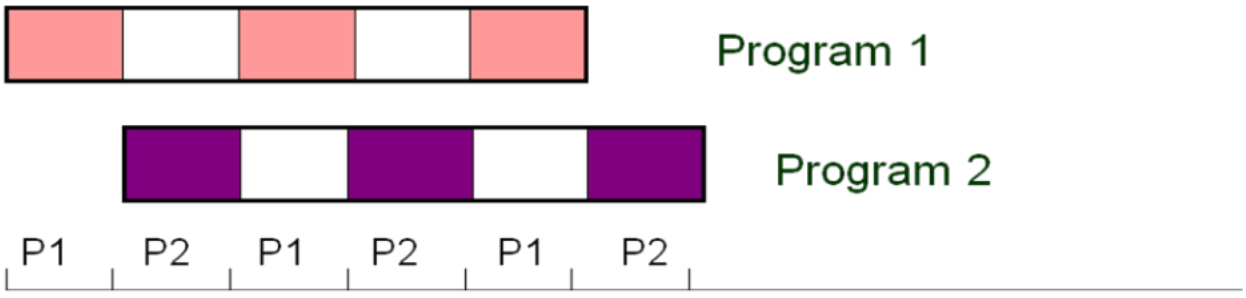
UNIT 2/ Module 2:Process Management
Chapter-5 : Process Scheduling

5.1 Basic Concepts

5.2 Scheduling Criteria

5.3 Scheduling Algorithms

5.1 Basic Concepts: Multiprogrammed execution

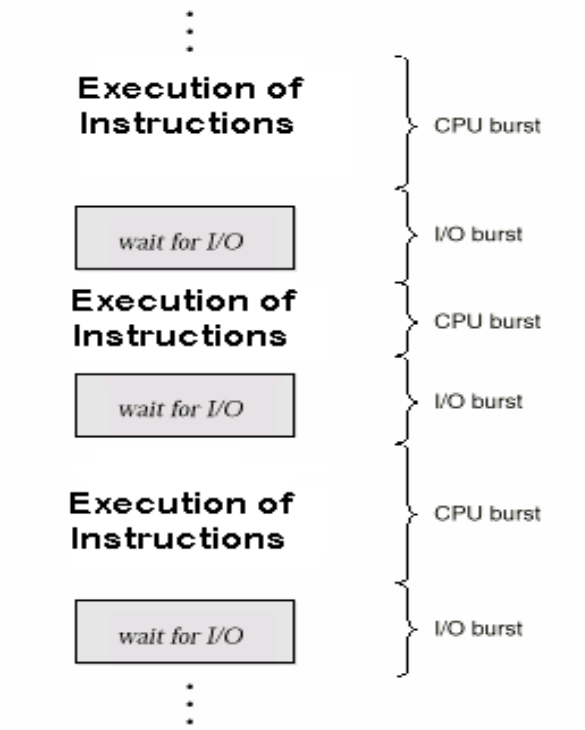


Basic Concepts

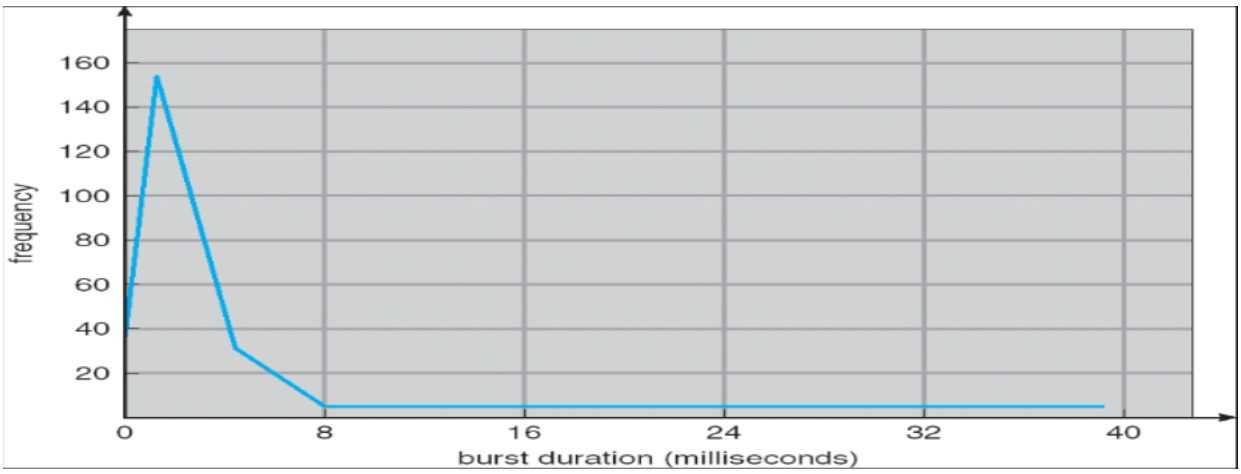
- How to obtain maximum CPU utilization with multiprogramming?
- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it.

Basic Concepts: CPU–I/O Burst Cycle

- CPU–I/O Burst Cycle
  - Process execution consists of a *cycle* of CPU execution and I/O wait.
- CPU bound process
  - Long CPU burst, Short I/O
- I/O bound process
  - Short CPU burst, Long I/O

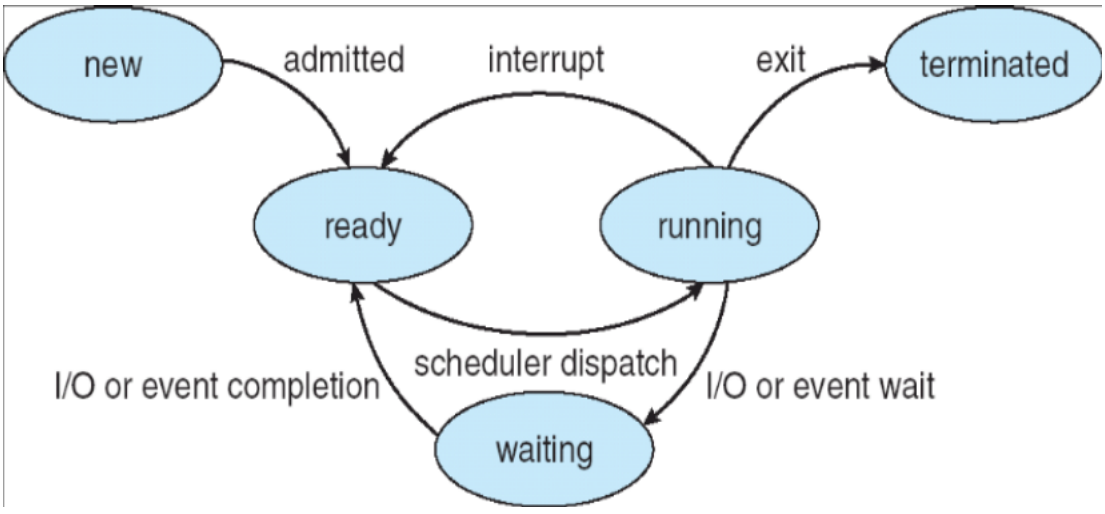


Basic Concepts: Histogram of CPU-burst Times



5.1.1 CPU Scheduler

- CPU Scheduler (short term scheduler) selects one process from the processes in memory that are ready to execute, and allocates the CPU.
- Ready queue may be implemented as FIFO queue, a tree or an ordered link where PCBs are linked waiting for CPU to be available.
- CPU scheduling decisions may take place when a process:
  - Switches from running to waiting state.
  - Switches from running to ready state.
  - Switches from waiting to ready.
  - Terminates.



5.1.2 Nonpreemptive

- No forceful switching from running state to waiting process.
- Once CPU has been allocated to a process, the process keeps the CPU, process switches from running state to waiting state only due to:
    - I/O request
    - Some OS service
    - When a process terminates by itself.

5.1.3 Preemptive

- Forceful switching from running state to waiting process.
    - Running process is interrupted and moved to waiting state by OS.
    - It may occur when a new process arrives.
- Cost is more in preemptive scheduling.

5.1.4 Preemptive & Nonpreemptive

- **Scheduling under *nonpreemptive***
    - When a process switches from running to waiting state.
    - When a process terminates.
    - All other scheduling is preemptive
    - When a process switches from running to ready state.
    - When a process switches from waiting to ready.
    - Windows 3.x use nonpreemptive, Windows 95 introduced preemptive.

5.1.5 Dispatcher

- Dispatcher module 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**
    - It is the time taken by the dispatcher to stop one process and start another running.

- Dispatcher should be very fast and dispatch latency should be less.

5.2 Scheduling Criteria

- Different CPU scheduling algorithms have different properties and choice of one algorithm depends on different criteria.
- CPU utilization – Aim is to keep the CPU as busy as possible (lightly loaded: 40%, heavily loaded: 90%)

  (High)
- Throughput – number of processes that complete their execution per unit time (very long process: 1 process/hour, short: 10 processes/sec)

  (High)
- Turnaround time – amount of time to execute a particular process (from submission to completion, includes execution time, I/O time and waiting time)

  (Least)
- Waiting time – amount of time a process has been waiting in the ready queue (CPU scheduling algorithm does not affect execution time and time required for I/O, it can only reduce waiting time)

  (Least)
- 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)

  (Least)

Optimization Criteria for scheduling

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

5.3 Scheduling Algorithms

- **First-Come, First-Served (FCFS)**
- **Shortest-Job-First (SJF)**

- **Priority Scheduling**
- **Round Robin (RR)**
- **Multilevel Queue Scheduling**
- **Multilevel Feedback Queue Scheduling**

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

- First request for CPU is served first.
- Implementation is managed by FIFO queue.
- New entered process is linked to tail of the queue.
- When CPU is free, it is allotted to the process at the head of the queue.

- **It is non-preemptive.**

Process	Burst Time (ms)
P_1	24
P_2	3
P_3	3

- **Suppose that the processes arrive in the order:**

  P_1 , P_2 , P_3

- **Gantt Chart for the schedule**

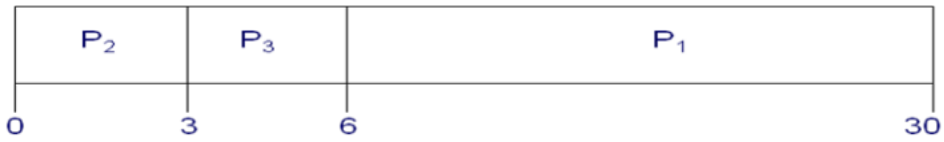
- Waiting time for P_1 = 0

  P_2 = 24

  P_3 = 27

- Average waiting time: (0 + 24 + 27)/3 = 51/3 = 17

- **Process** **Burst Time**
- P_1 24
- P_2 3
- P_3 3
- Suppose that the processes arrive in the order
- P_2 , P_3 , P_1 .
- **The Gantt chart for the schedule**



- Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 9/3 = 3
- Much better than previous case.

Convoy effect

- Thus average waiting time in FCFS vary substantially if CPU burst time vary greatly.
- *Convoy effect* is due to short processes behind a long process.
- All the processes wait for the one big process to get off the CPU.
- It results in lower CPU and device utilization.
- It is not good for time-sharing system.

5.3.2 Shortest-Job-First (SJF) Scheduling

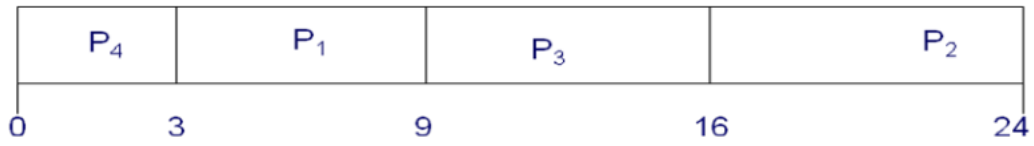
- SJF associates with each process the length of its next CPU burst (Shortest-next-CPU burst algorithm)
- CPU is assigned to the process with the shortest next CPU burst time.
- If two processes have the same length CPU burst, FCFS is used to break the tie.
- SJF is optimal and gives minimum average waiting time for a given set of processes.

Example of SJF

Process **Burst Time**

P_1	6
P_2	8
P_3	7
P_4	3

- **Gantt Chart for the schedule**



Waiting for each process

- P1 →3, P2 →16, P3 →9, P4 →0
- Average waiting time = (3 + 16 + 9 + 0)/4 = 28/4 = 7 ms

Shortest-Job-First (SJF) Scheduling

Two schemes:

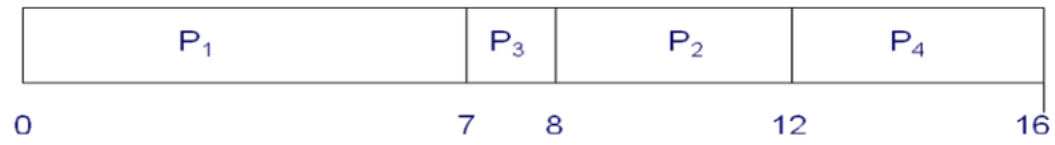
- **Nonpreemptive SJF**
  - Once CPU is given to the process it cannot be preempted until completes its CPU burst.
  - Preemptive SJF
  - If a new process arrives with CPU burst length less than remaining time of current executing process, current process is preempted. This scheme is know as
  - Shortest-Remaining-Time-First (SRTF).

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1

P_4	5.0	4

- **Gantt Chart for the schedule**



- Average waiting time = (0 + (7-4) + (8-2) + (12-5))/4 = 16/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- **Gantt Chart for the schedule**

- Average waiting time = (9 + 1 + 0 +2)/4 = 12/4 = 3ms

5.3.3 Priority Scheduling

- A priority number (integer) is associated with each process.
- Priorities can be defined internally or externally.
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
- Equal priority processes are scheduled in FCFS order.

Priority Scheduling (nonpreemptive)

- New process having the highest priority comes at head of the ready queue.
- Priority scheduling can be:
  - Preemptive

- Nonpreemptive

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

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

- The Gantt chart for the schedule is:
- Average waiting time= (6+0+16+18+1)/5 = 41/5 = 8.2
- Priority of newly arrived process is compared with already running process.
- Scheduler will preempt the CPU if the priority of newly arrived process is higher than priority of running process.

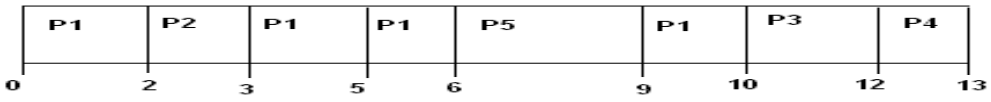
Process	Burst Time	Priority	Arrival Time
P_1	6	3	0
P_2	1	1	2
P_3	2	4	3
P_4	1	5	5

❖ P_5 3 2 6

Process	Burst Time	Priority	Arrival Time
P_1	6	3	0
P_2	1	1	2
P_3	2	4	3
P_4	1	5	5

P_5 3 2 6

❖ **The Gantt chart for the schedule is:**



❖ **Average waiting time= (4+0+7+7+0)/5 = 18/5 = 3.6**

Priority Scheduling

❖ Problem is Starvation i.e. low priority processes may wait for indefinite time or never execute.

(In a heavily loaded system, a steady stream of higher priority process can prevent a low priority process from ever getting the CPU)

❖ Solution of starvation is Aging i.e. technique of gradually increasing the priority of processes that wait in the system for a long time.

(e.g. If priorities range is 100(lowest) to 0(highest), process with 100 priority is decremented by one after each 10 minutes and at last process will gain 0, highest, priority)

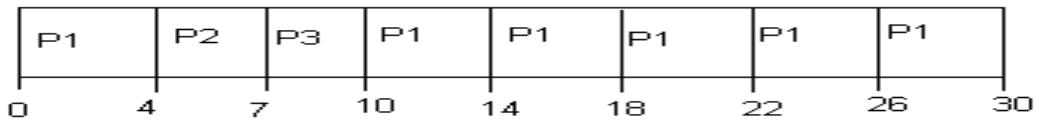
5.3.5 Round Robin (RR) Scheduling

- It is preemptive scheduling & designed for time-sharing system.
- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds.
- After this time elapses, the process is preempted and added to the end of the ready queue.
- CPU scheduler goes around the ready queue allocating the CPU to each process for a time interval of up to 1 time quantum.
- Implementation is through FIFO queue of processes.
- New processes are added to the tail of the ready queue.
- CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum and dispatches the process.
- After 1 time quantum, process either will finish the execution or preempted and scheduler will schedule other process from head of the queue.

Example of RR with Time Quantum = 4

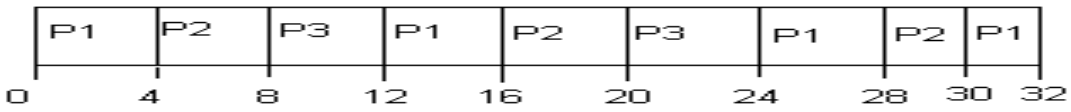
Process	Burst Time
P_1	24
P_2	3
P_3	3

- **The Gantt chart is:**



- **Average waiting time = ((10-4)+4+7)/3 = 5.67**
Process	Burst Time
P_1	14
P_2	10
P_3	8

- **The Gantt chart is:**



- **Average waiting time = (18+20+16)/3 = 18**

Round Robin (RR) Scheduling

- **Performance of RR depends on the size of time quantum.**
  - *q (time quantum)* is large, RR works similar to FCFS
  - *q (time quantum)* is small, overhead due to context switch is too high
  - *time quantum should be optimum.*

Example: (different time quantum)**One process with process time 10**

……..