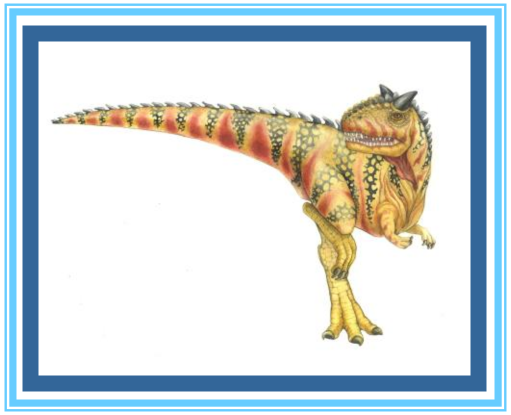
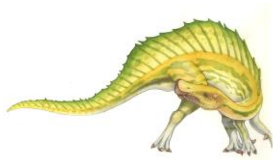
Chapter 5: Process Scheduling



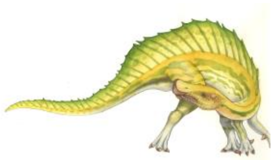
Chapter 6: Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling



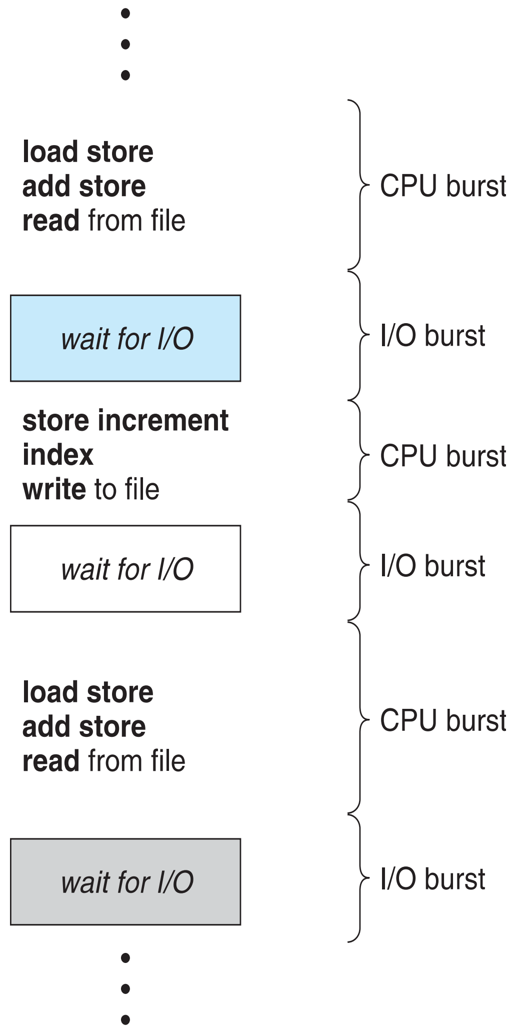
Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To understand the scheduling algorithms of operating systems

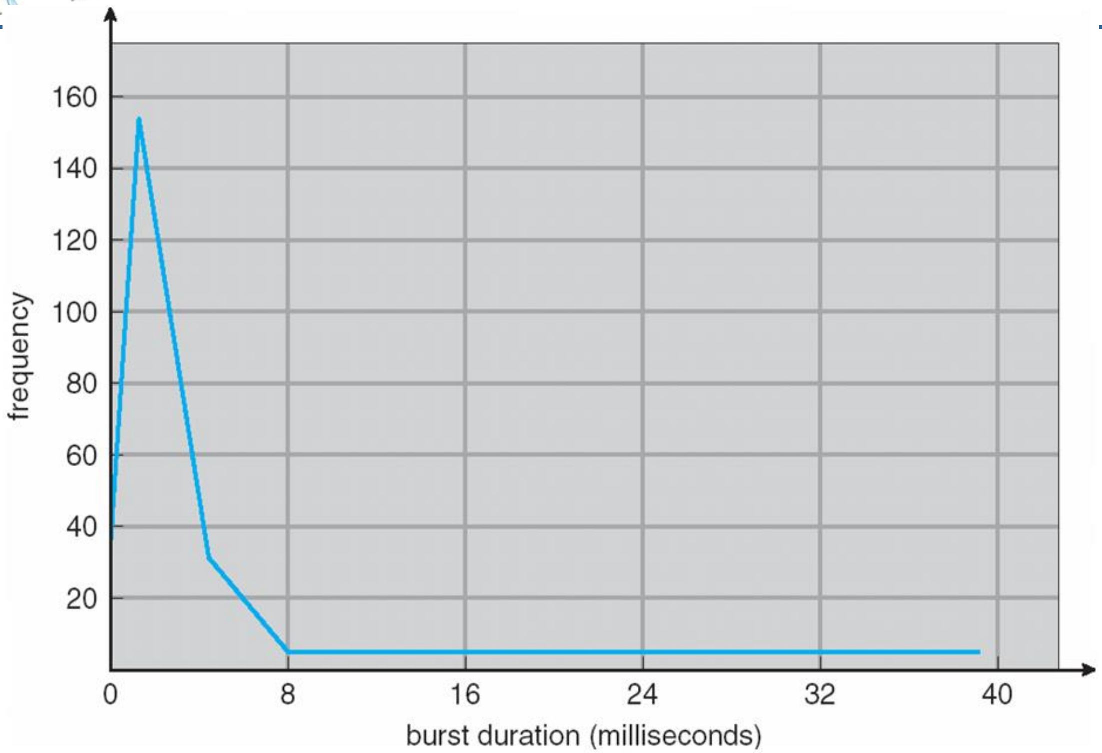


Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Histogram of CPU-burst Times



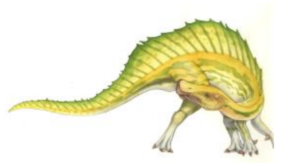
- The curve is generally characterized as **exponential or hyperexponential**, with a large number of short CPU bursts and a small number of long CPU bursts
- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.

algorithm.

- This distribution is important for selecting CPU-scheduling alg

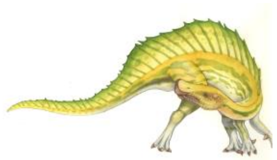
CPU Scheduler

- **Short-term scheduler** selects from among the processes in **ready queue**, and allocates the **CPU** to one of them
  - Ready Queue may be ordered in various ways(FIFO, priority etc)
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive or cooperative**
- All other scheduling is **preemptive**
  - Consider access to shared data (may result in inconsistent data)
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities



Dispatcher

- Dispatcher is another module involved n CPU scheduling.
- It 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** – time it takes for the dispatcher to stop one process and start another running

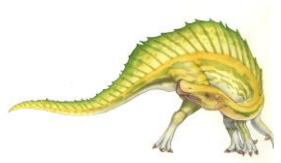


Scheduling Criteria

Criteria for comparing CPU-scheduling algorithm

- **CPU utilization** – Must keep the CPU as busy as **possible(40% to 90% utilization)**
- **Throughput** – # of processes that complete their execution per time unit. For long process it may be 1 process/hour and for short process it may be 10 process/sec
- **Turnaround time** – The interval from the time of submission of a process to the time of completion. 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** – amount of time a process has been 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)

It is desirable to **maximize** **CPU utilization and throughput** and to **minimize** **turnaround time, waiting time, and response time**.

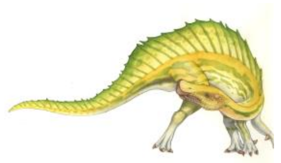


Scheduling Algorithm Optimization Criteria

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

- TAT = CT – AT
- WT = TAT – BT

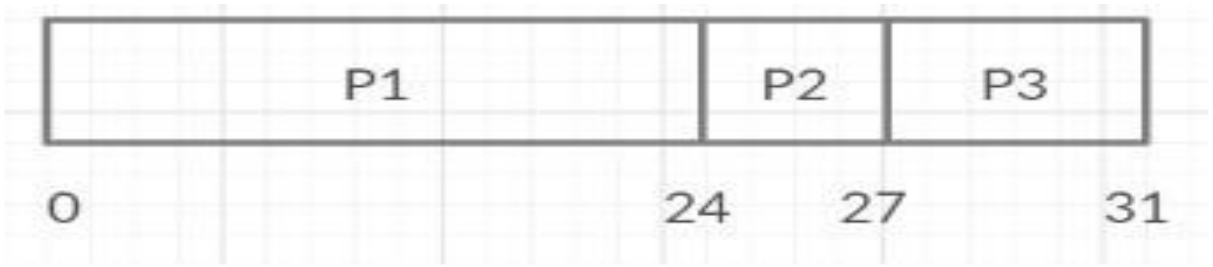
- **Completion Time (CT), . Arrival Time (AT),**
- **Waiting Time (WT) , Burst Time (BT):**



Example:

Process	Burst Time (in sec.)
P1	24
P2	3
P3	4

- Gantt Chart:



- Avg. TAT = (24 + 27 + 31) / 3 = 27.33 sec
- Avg. WT = (0 + 24 + 27) / 3 = 17.0 sec

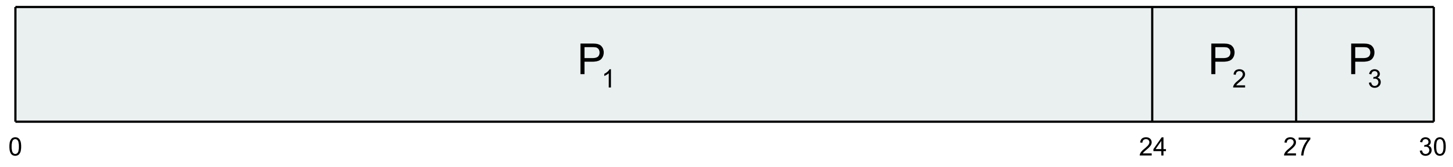
First- Come, First-Served (FCFS) Scheduling

- Simplest CPU-scheduling algorithm
- 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.
- On the negative side, the average waiting time under the FCFS policy is often quite long.
- Consider the example with set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



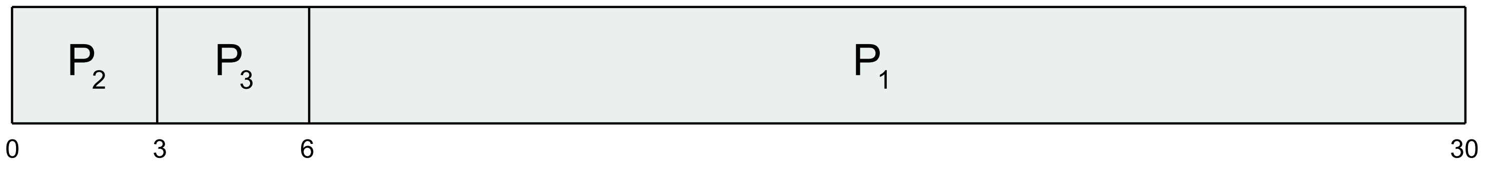
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

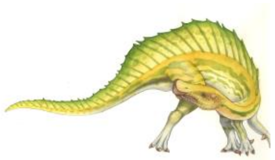
- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- **Convoy effect** - short process wait for one long process to get the CPU
- This result in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.
  - Consider one long CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

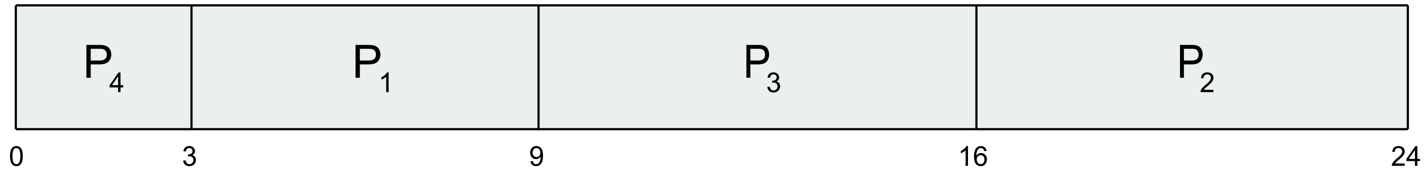
- Associates with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - SJF is used frequently **in long term scheduler**, with user specifying the process time during submission.
  - Difficult to implement at s**hort-term scheduler** level.
  - For short-term scheduler next CPU burst can be calculated or predicted using the current CPU burst.



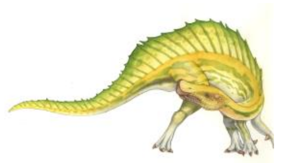
Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7



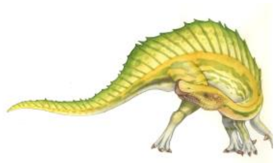
Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

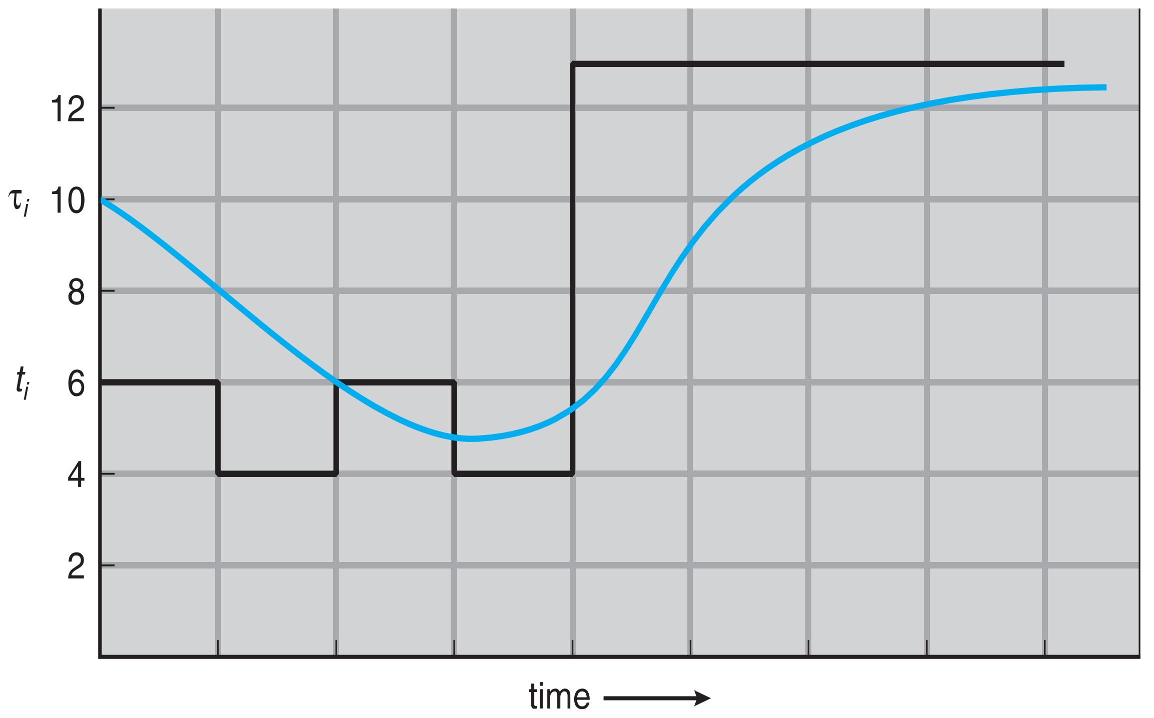
- Can be done by using the length of previous CPU bursts, using **exponential averaging**

  1. $t_n =$ actual length of n^{th} CPU burst
  2. $\tau_{n+1} =$ predicted value for the next CPU burst
  3. $\alpha, 0 \le \alpha \le 1$
  4. Define: $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n.$

- α controls the relative weight of recent and past history in our prediction
- Commonly, α set to ½
- If α = 1, then τ_{n+1} = t_n, and only the most recent CPU burst matters.
- Preemptive version called **shortest-remaining-time-first**



Prediction of the Length of the Next CPU Burst



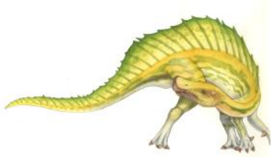
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots + (1 - \alpha)^j \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

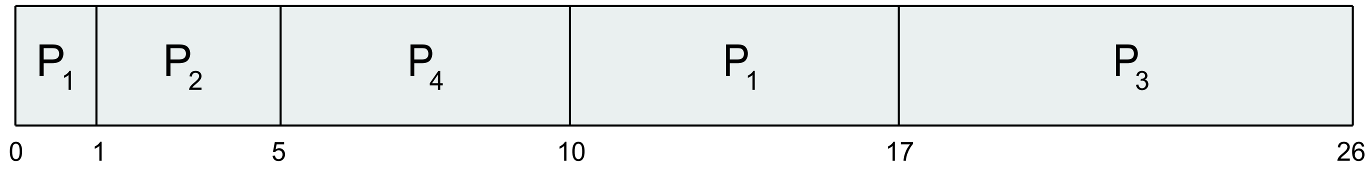


Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

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

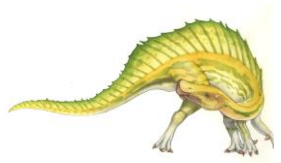
- ***Preemptive* SJF Gantt Chart**



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec
- What is the average waiting time of NP-SJF ?

Priority Scheduling

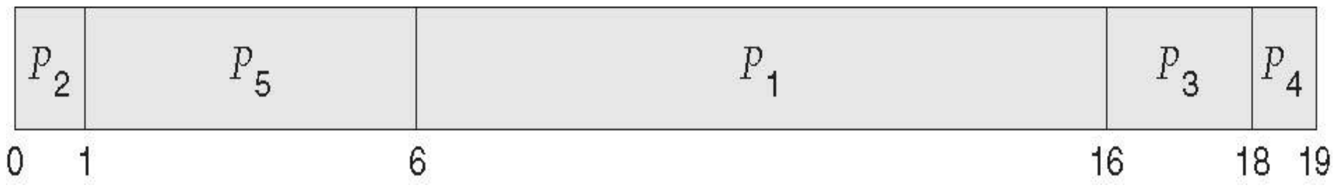
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process



Example of 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

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

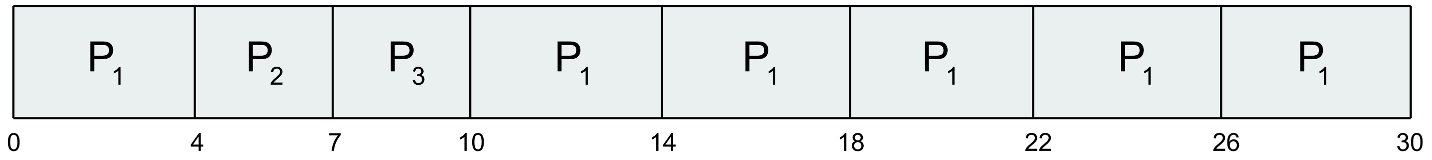
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- 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*-1)*q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

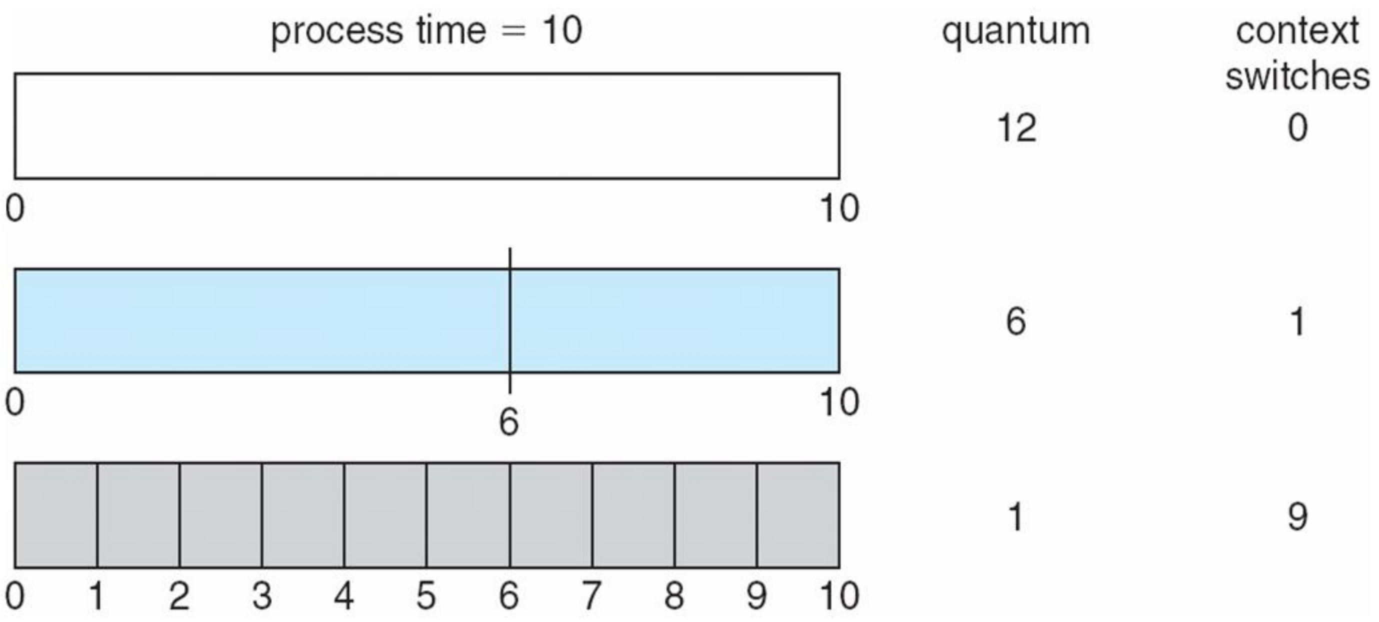
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

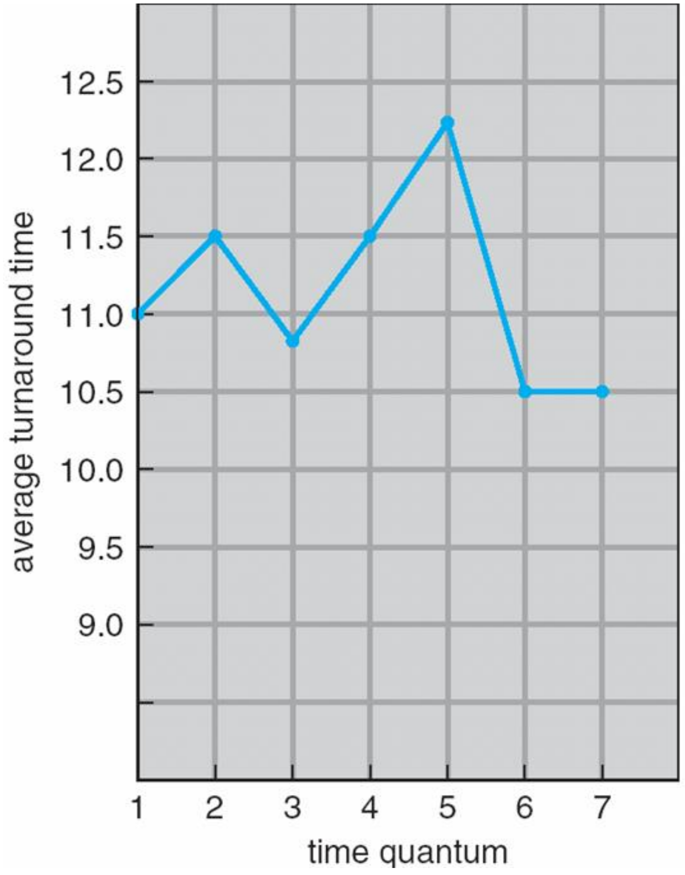


- **Process waits for, P1=(10-4)=6, P2=4, P3=7, average waiting time is 17/3=5.66 ms**
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



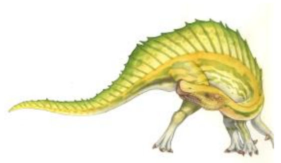
process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

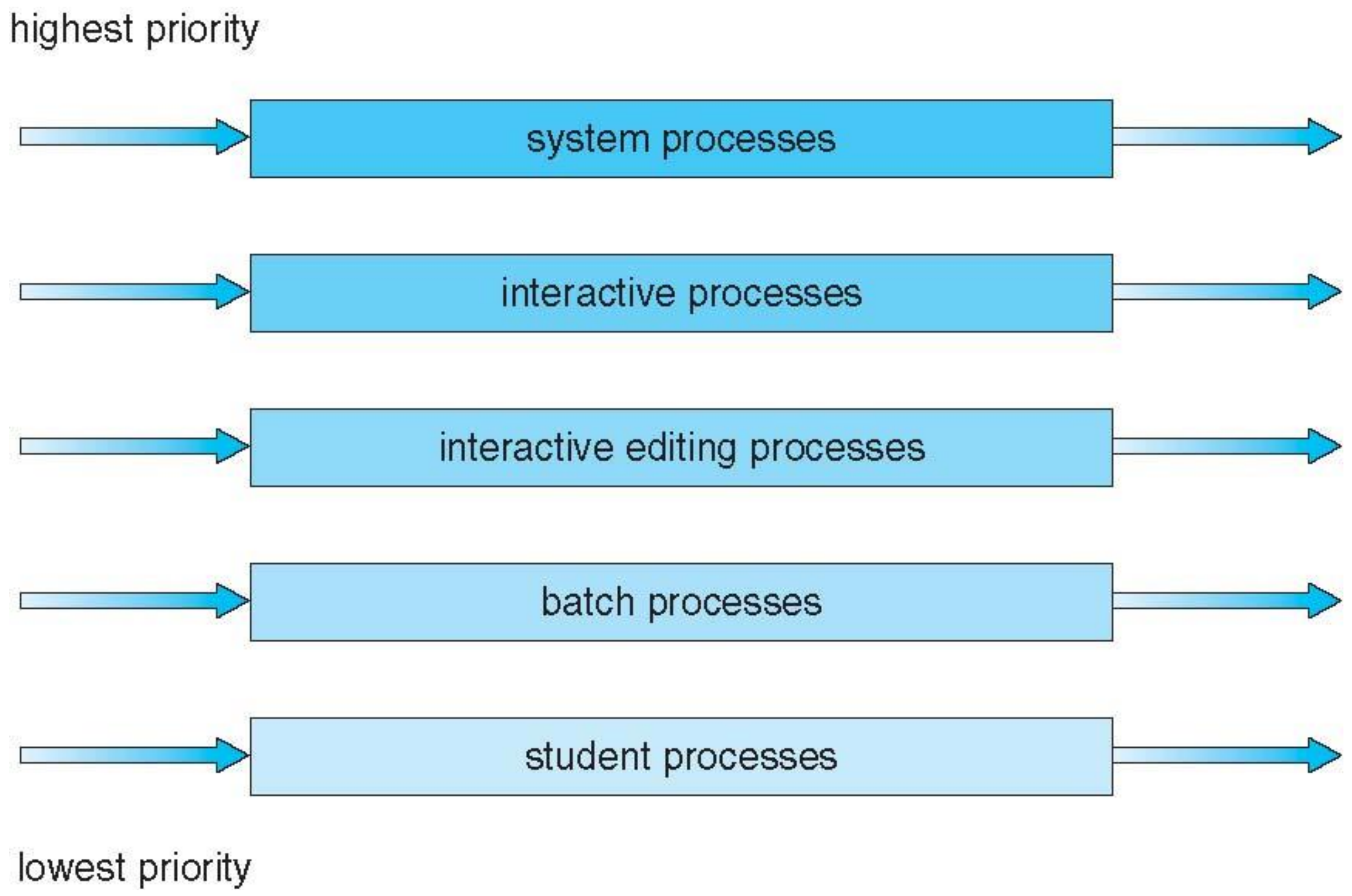
- Turn around time also depends upon the time quantum
- But, average turn around time does not necessarily improve with increase in time quantum, as shown in fig.
- Generally, average TAT can improve if most processes finish their next CPU burst in a single quantum

Multilevel Queue

- **Ready queue** is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently assigned a queue based upon properties like **memory size, process type, priority etc**.
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

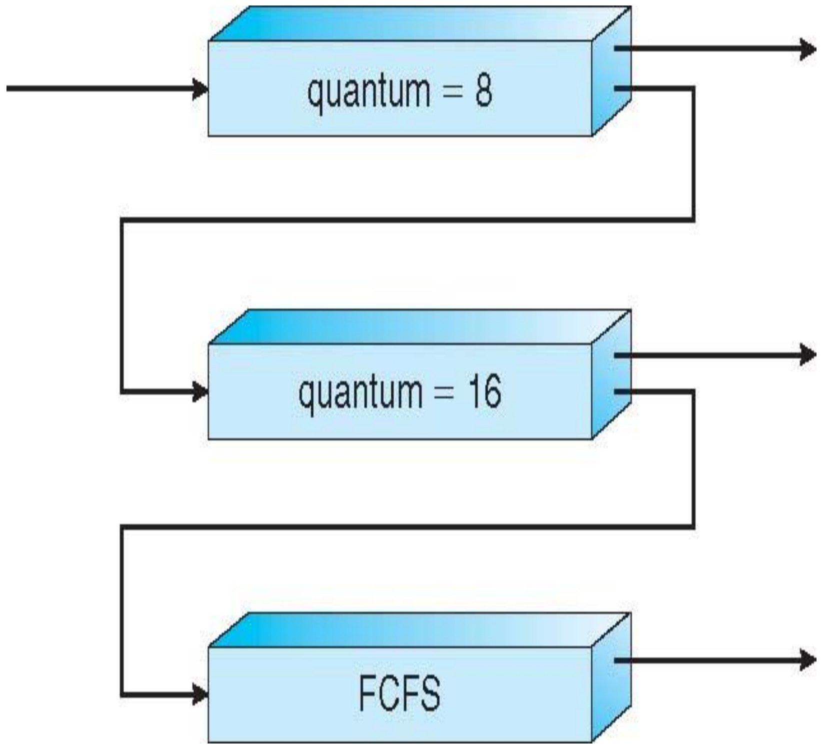


Multilevel Queue Scheduling



Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Multilevel Feedback Queue

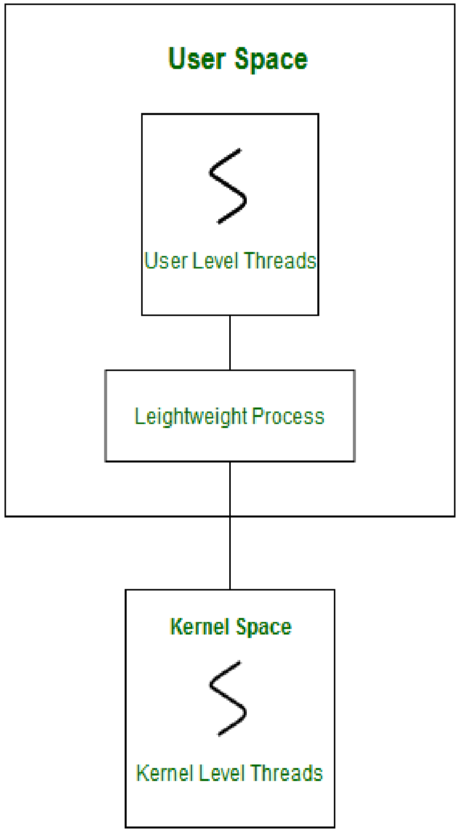
- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Thread Scheduling

- Scheduling of threads involves two boundary scheduling.
- Scheduling of **user-level threads (ULT**) to kernel-level **threads (KLT)** via lightweight process (LWP)
- Scheduling of **kernel-level threads** by the system scheduler to perform different unique OS functions.

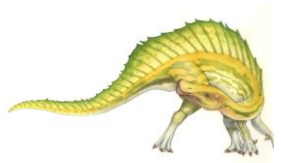
Lightweight Process (LWP)

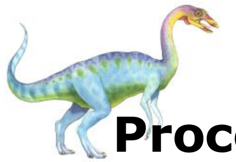
- Light-weight process are threads in the user space that acts as an interface for the ULT to access the physical CPU resources.
- Thread library schedules which thread of a process to run on which LWP and how long.
- The number of LWPs created by the thread library depends on the type of application.



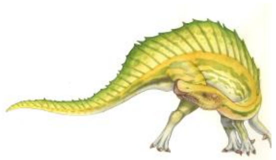
Thread Scheduling

- **Contention Scope:** The word contention here refers to the competition among the User level threads to access the kernel resources. Thus, this control defines the extent to which contention takes place. It is defined by the application developer using the thread library.
- Contention Scope is classified as-
- .**Process Contention Scope (PCS) :**
  The contention takes place among threads **within a same process**. The thread library schedules the high-prioritized PCS thread to access the resources via available LWPs (priority as specified by the application developer during thread creation).
- **System Contention Scope (SCS) :**
  The contention takes place among **all threads in the system**. In this case, every SCS thread is associated to each LWP by the thread library and are scheduled by the system scheduler to access the kernel resources.





Process Contention Scope (PCS) refers to the scope within which threads compete for CPU resources in a multithreaded environment. It determines how threads are scheduled and managed in relation to each other when they belong to the same process.
In PCS, threads compete only with other threads within the **same process** for CPU time.
The thread scheduler responsible for PCS is usually part of the **user-level thread library**.
PCS is also called **user-level scheduling** or **user-level contention scope**.
Since the operating system sees the entire process as a single entity (one or more kernel threads), it only schedules the process, not the individual threads.
Within the process, the user-level thread scheduler manages thread execution, switching threads according to PCS rules.
PCS provides faster thread switching and scheduling because it avoids kernel mode transitions.
However, a major drawback is that if one thread makes a blocking system call, the entire process may be blocked because the kernel is unaware of individual threads.

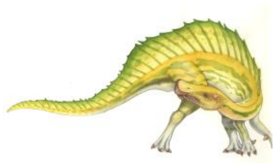


Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
- The POSIX Pthread library provides function ***Pthread_attr_setscope*** to define the type of contention scope for a thread during its creation.

  int Pthread_attr_setscope(pthread_attr_t *attr, int scope)
- The first parameter denotes to **which thread within the process** the scope is defined.
- The second parameter defines **the scope of contention** for the thread pointed. It takes two values.
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling

System Contention Scope (SCS) defines how threads contend for CPU resources across the entire system, meaning threads compete with **all other threads from all processes** for CPU time.
In SCS, the thread scheduler is part of the **operating system kernel**.
Threads are mapped to kernel-level threads, and the kernel schedules them directly.
This allows the operating system to manage thread execution across all processes on the system.
Since the kernel schedules threads system-wide, threads from different processes compete fairly for CPU time.
SCS enables true parallelism on multiprocessor systems by distributing threads across multiple CPUs.
It also handles blocking system calls better because other threads in the same process or different processes can continue to run.
However, scheduling at the kernel level involves more overhead than user-level scheduling, causing potentially slower thread management.



Exercises

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

Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of these scheduling algorithms?

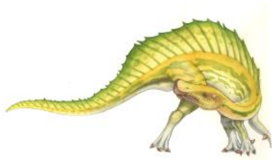
d. Which of the algorithms results in the minimum average waiting time (over all processes)?

Exercise 3

Consider the following set of processes, arriving at the given times and having the following CPU burst time and priorities (Smaller number is having higher priority):

Draw a Gantt chart and calculate average waiting time and turnaround time of each process using SJF, Priority and Round robin (quantum 3 ms). Assume pre-emptive scheduling policy for SJF and Priority scheduling.

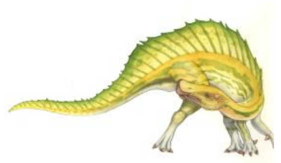
Process	Arrival Time(ms)	Burst Time (ms)	Priority
A	0	8	3
B	3	4	1
C	5	7	4
D	8	3	2



Exercises

The following processes are being scheduled using a preemptive, round-robin scheduling algorithm. Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an ***idle task*** (which consumes no CPU resources and is identified as P_{idle}). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

Thread	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105



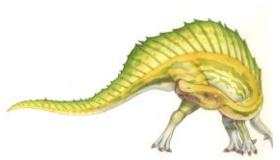
Exercises

a. Show the scheduling order of the processes using a Gantt chart.

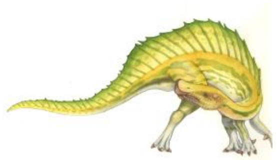
b. What is the turnaround time for each process?

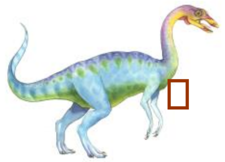
c. What is the waiting time for each process?

d. What is the CPU utilization rate?

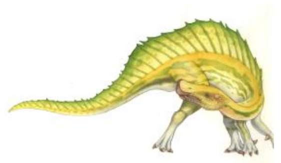


- The **nice value** in Linux is a user-space concept used to influence the **priority** of a process in CPU scheduling. It helps determine how much CPU time a process should get relative to other processes.
- The nice value ranges from **-20 to +19**:
  - **-20** means the highest priority (least "nice" to other processes, so it gets more CPU time).
  - **+19** means the lowest priority (most "nice" to other processes, so it gets less CPU time).
- By default, processes start with a nice value of **0**.
- A process with a lower nice value (higher priority) will be favored by the scheduler and get more CPU resources.
- Conversely, a process with a higher nice value (lower priority) will be scheduled less often.





- Both **Active** and **Expire arrays** are **process queues** (usually implemented as arrays or lists) that hold processes that are ready to run.
- Each array contains processes organized by priority.
- **How do they work?**
- **Active Array:**
  - Holds all processes that are currently eligible for CPU time.
  - The scheduler picks the next process to run from the active array.
- **Expire Array:**
  - Holds processes that have exhausted their current time slice (quantum).
  - Once a process in the active array uses up its allocated CPU time, it is moved to the expire array.
- **Switching:**
  - When the active array becomes empty (all processes have used their time slice), the scheduler **swaps** the active and expire arrays.
  - This means the expire array becomes the new active array, and those processes get their next turn for CPU time.
  - The old active array becomes the new expire array.



End of Chapter 5