Chapter: CPU/ Process Scheduling

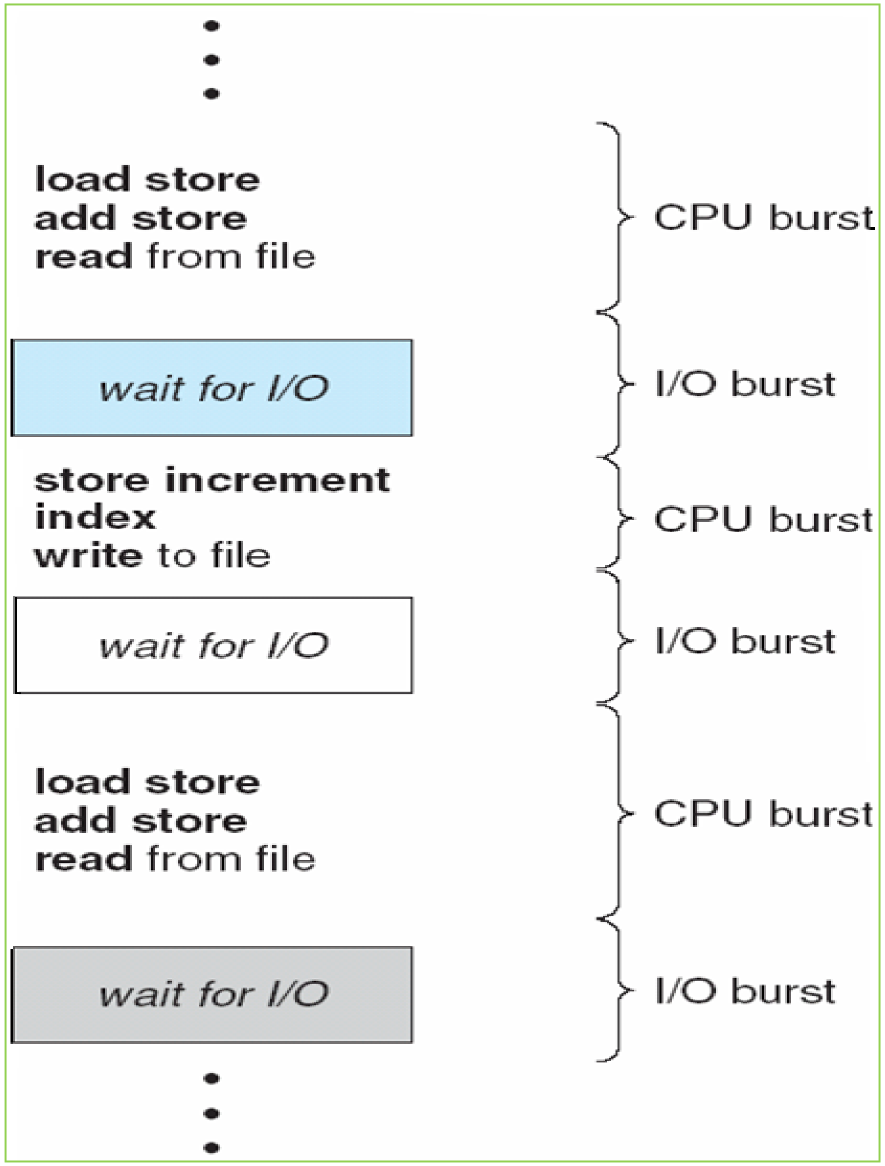
Basic Concept

- CPU Scheduling is basis of Multi-programmed OS
- Objective of Multi-programming is to have some processes running all the time to maximize CPU Utilization.

Sequence of CPU and I/O Bursts

- **CPU Burst-** when process is executed in CPU
- **CPU Burst Time:** The amount of **time** the **process uses the processor**
- Types of **CPU bursts**:
  - Long **bursts** -- process is **CPU** bound (spend max. time with CPU
  - Short **bursts** -- process is I/O bound (spend max. time with I/O)
- **I/O Burst-** when CPU is waiting for I/O completion for further execution.

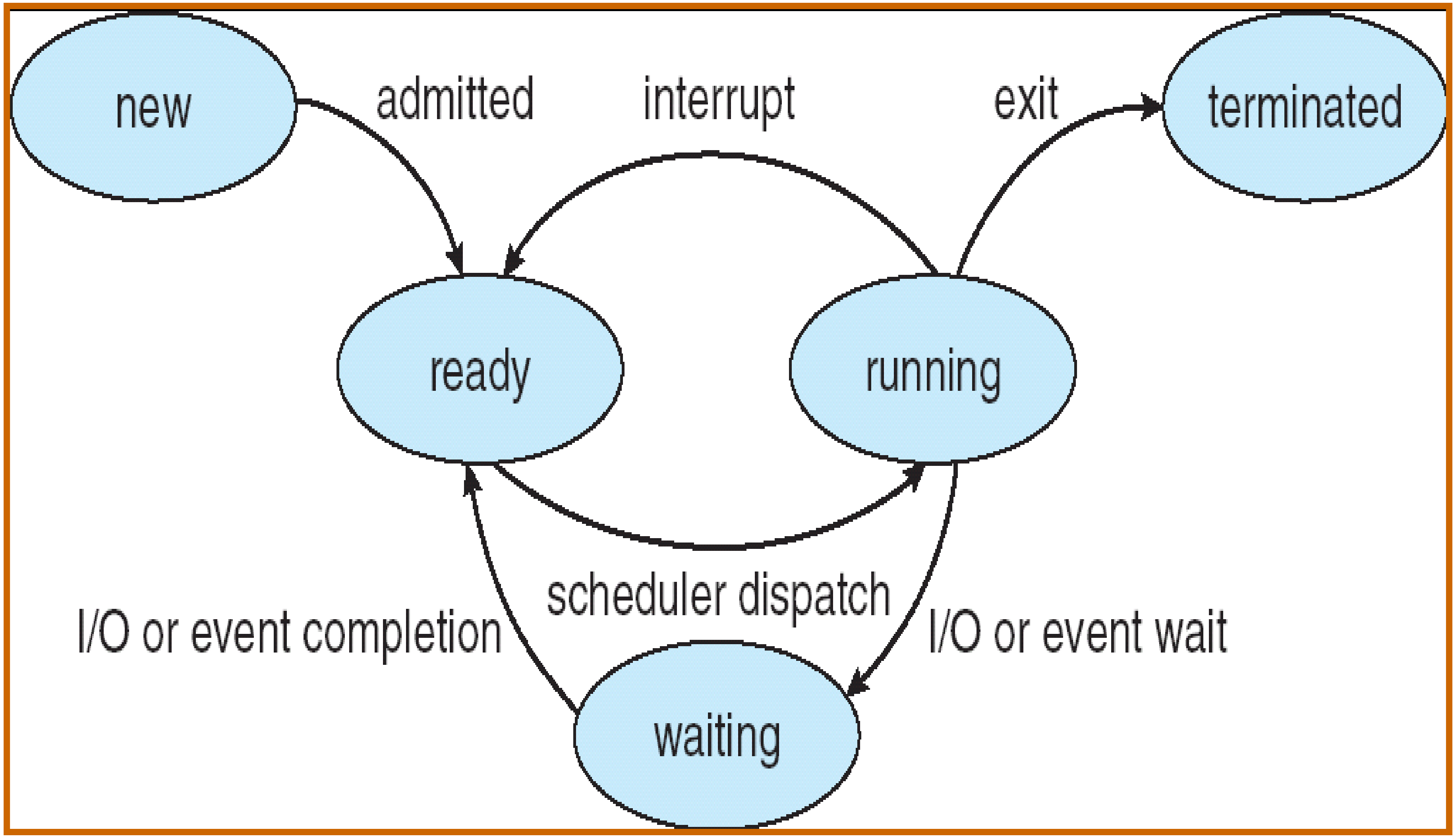
Sequence of CPU and I/O Bursts



CPU Scheduling Decisions

1. When process switches from Running State to Waiting State (i/o request or wait)	**1. Non-preemptive**
2. When process switches from Running to Ready State (interrupt)	**2. Pre-emptive**
3. When process switches from Waiting State to Ready State (at completion of i/o)	**3. Pre-emptive**
4. When a process terminates	**4. Non-Preemptive (allow)**

Process States



Scheduling

- Non-Preemptive
- Preemptive

Non-Preemptive or Cooperative Scheduling

- Once the CPU is allocated to a process, process keeps the CPU until:
  - it releases when it completes
  - by switching to waiting state

  E.g : 1. Windows 3.x and Apple Macintosh operating systems uses non-preemptive scheduling

  2. Windows (also 10) uses a round-robin technique with a *multi-level feedback queue* for priority scheduling
- Process is executed till completion. It cannot be interrupted.
  Eg First In First Out

Preemptive Scheduling

- The running process is interrupted for some time and resumed later on, when the priority task has finished its execution.

- CPU /resources is/are taken away from the process when some high priority process needs execution.

Dispatcher

- A module that gives control of CPU to the process selected by short-term scheduler.
- Functions:
  - Switching Context
  - Switching to User mode
  - Jumping to proper location to restart the program.
- The dispatcher should be as fast as possible, given that it is invoked during every process switch
- Dispatch Latency:
  - Time taken for the dispatcher **to stop one process and start another running.**

Scheduling Criteria

- Which algorithm to use in a particular situation

1. CPU Utilization: CPU should be busy to the fullest

2. Throughput: No. of processes completed per unit of time.

Turnaround Time: The time interval from submitting a process to the time of completion.

The time interval from submitting a process to the time of completion

Turnaround Time= Time spent to get into memory + waiting in ready queue + doing I/O + executing on CPU

(It is the amount of time taken to execute a particular process)

Scheduling Criteria

4. **Waiting Time:** Time a process spends in ready queue.

Amount of time a process has been waiting in the ready queue to acquire control on the CPU.

5. **Response Time:** Time from the submission of a request to the first response, Not Output

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

Scheduling Algorithm Optimization Criteria

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

First-Come, First-Served (FCFS)

- Processes that request CPU first, are allocated the CPU first

- FCFS is implemented with FIFO queue.

- A process is allocated the CPU according to their arrival times.

- When process enters the ready queue, its PCB is attached to the Tail of queue, When CPU is free, it is allocated to the process selected from Head/Front of queue.

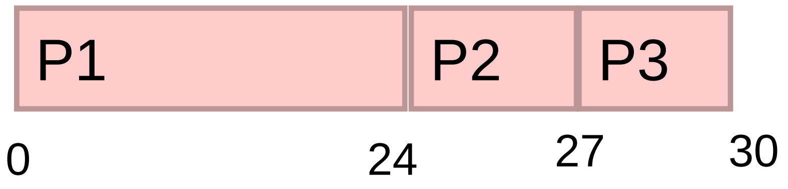
First-Come, First-Served (FCFS)

- “Run until Completed:” FIFO algorithm
- Example: Three processes arrive in order P1, P2, P3.
  - P1 burst time: 24
  - P2 burst time: 3
  - P3 burst time: 3
- Draw the Gantt Chart and compute Average Waiting Time and Average Turn Around Time/Completion Time.

Sol: As arrival time is not given assume order of arrival as: P1,P2,P3

First-Come, First-Served (FCFS)

- Example: Three processes arrive in order P1, P2, P3.
  - P1 burst time: 24
  - P2 burst time: 3
  - P3 burst time: 3
- Waiting Time
  - P1: 0
  - P2: 24
  - P3: 27



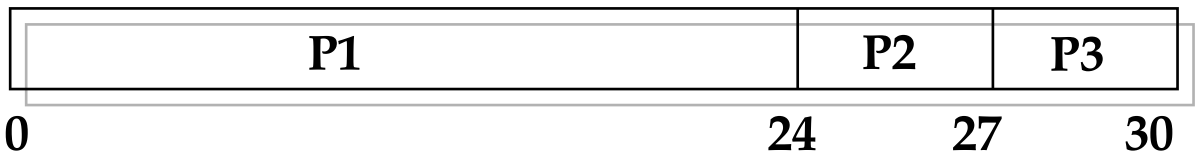
- Turnaround Time/Completion Time:
  - P1: 24
  - P2: 27
  - P3: 30

- Average Waiting Time: (0+24+27)/3 = (51/3)= 17 milliseconds
- Average Completion Time: (24+27+30)/3 = 81/3=27 milliseconds

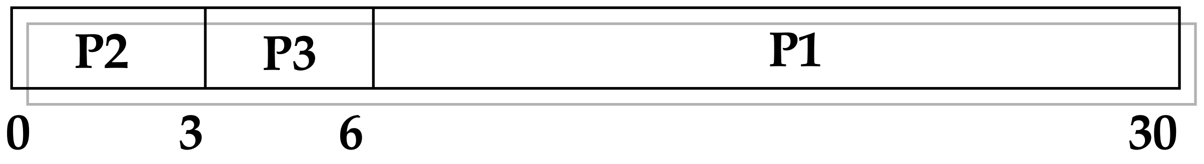
First-Come, First-Served Scheduling (1)

- Simplest, non-preemptive, often having a long average waiting time
- Example:

Process	Burst Time
P1	24
P2	3
P3	3



AWT = (0+24+27)/3
= 17

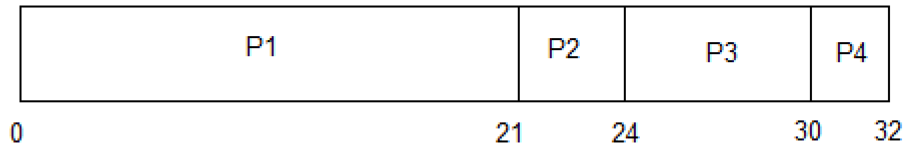


AWT = (6+0+3)/3
= 3

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

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms



This is the GANTT chart for the above processes

Shortest Job First

- Processes with least execution time are selected first.
- CPU is assigned to process with less CPU burst time.
- SJF:
  - Non-Preemption: CPU is always allocated to the process with least burst time and Process Keeps CPU with it until it is completed.
  - Pre-Emption: When a new process enters the queue, scheduler **checks its execution time and compare with the already running process**.
  - If Execution time of running process is more, CPU is taken from it and given to new process.

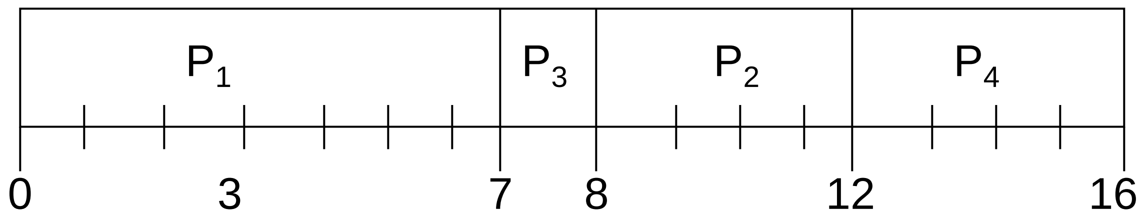
Shortest-Job First (SJF) Scheduling

- Two schemes
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the **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

- SJF (non-preemptive)



- Average waiting time = [0 + (8-2) + 3 + 7]/4 = 4

Shortest Job First(Non-Preemptive)

Q1. Consider foll. Processes with A.T and B.T

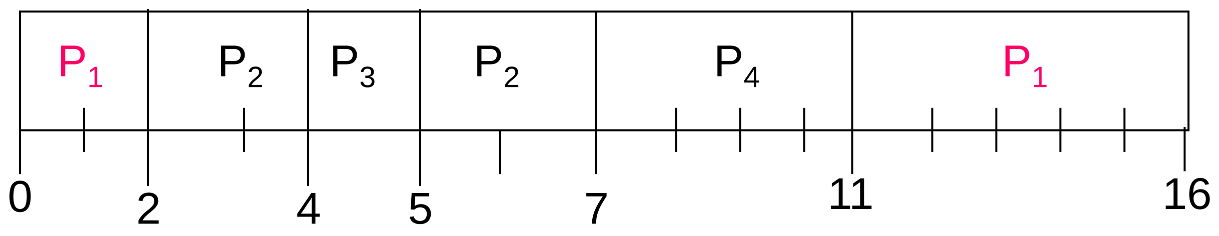
Process	A.T	B.T
P1	1	7
P2	2	5
P3	3	1
P4	4	2
P5	5	8

Cal. Completion time, turn around time and avg. waiting time.

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

- SJF (preemptive)



- Average waiting time = [(16-7) + 1 + 0 +2]/4 = 3

Shortest Job First(Preemptive)

Q1. Consider foll. Processes with A.T and B.T

Process	A.T	B.T
P1	0	9
P2	1	4
P3	2	9

Cal. Completion time, turn around time and avg. waiting time.

SJF(Pre-emptive)-> SRTF

Shortest Job First(Preemptive)

Q1. Consider foll. Processes with A.T and B.T

Process	A.T	B.T
P1	0	5
P2	1	3
P3	2	3
P4	3	1

Cal. Completion time, turn around time and avg. waiting time.

Practice: Shortest Job First (Non Preemption)

- P1 burst time: 15
- P2 burst time: 8
- P3 burst time: 10
- P4 burst time: 3

Practice: Shortest Job First (Preemption)

Process	Arrival Time	Burst Time			
P1	0	8			
P2	1	4			
P3	2	9			
P4	3	5			

Practice: Shortest Job First (Preemption)

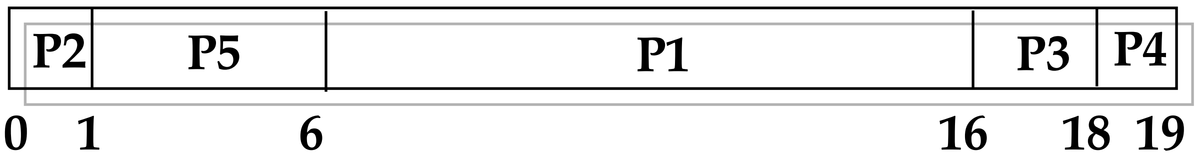
Process	Arrival Time	Burst Time	CT	Turn around time	Waiting Time
P1	0	7	16		
P2	2	4	7		
P3	4	1	5		
P4	5	4	11		

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 a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute. (At MIT, there was a job submitted in 1967 that had not be run in 1973.)
- Solution ≡ **Aging** – as time progresses increase the priority of the process

An Example

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



AWT = (6+0+16+18+1)/5=8.2

Priority Scheduling (Preemptive)

Process	Arrival Time	Priority	Burst Time	Completion Time
P1	1	5	4	
P2	2	7	2	
P3	3	4	3	

Consider 4 as Highest and 7 as Lowest Priority

Priority Scheduling (Preemptive)

Process	Arrival Time	Priority	Burst Time	Completion Time
P1	0	2	10	
P2	2	1	5	
P3	3	0	2	
P4	5	3	20	

Consider 3 as Lowest and 0 as Highest Priority

Priority Scheduling (Preemptive)

Process	Arrival Time	Priority	Burst Time	Completion Time
P1	1	4	4	
P2	2	5	2	
P3	2	7	3	
P4	3	8	5	
P5	3	5	1	
P6	4	6	2	

Consider 4 as Lowest and 8 as Highest Priority

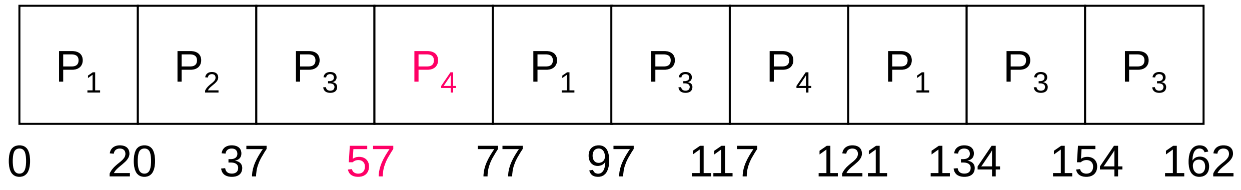
Round Robin Scheduling

- A Time Quantum is associated to all processes
- Time Quantum: Maximum amount of time for which process can run once it is scheduled.
- RR scheduling is always Pre-emptive.

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better *response*

Round Robin

TQ: 2

Process	Arrival Time	Burst Time	Completion Time
P1	0	5	10
P2	1	7	13
P3	2	1	5

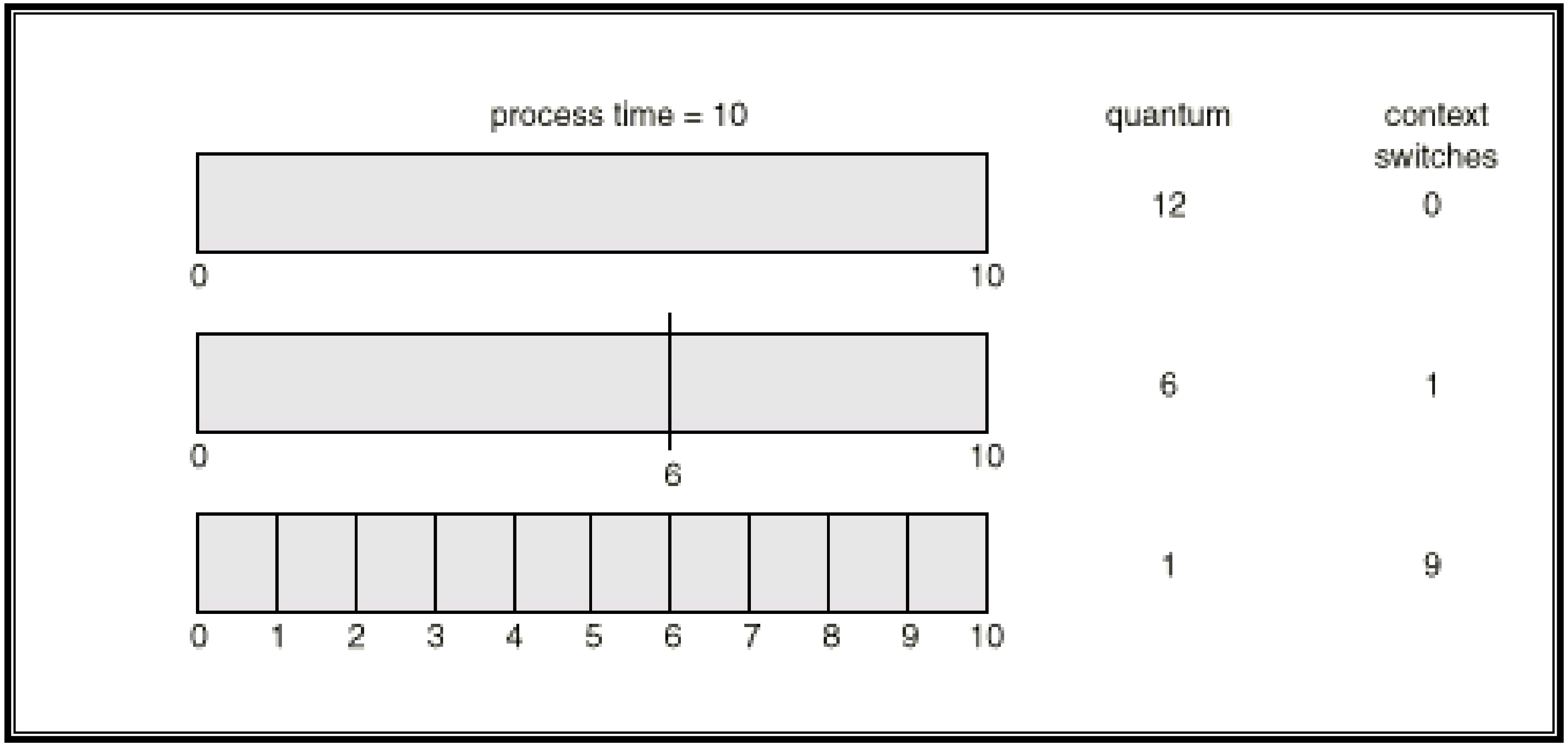
Process	Arrival Time	Burst Time	Completion Time
P1	0	3	
P2	3	4	
P3	4	6	

Round Robin

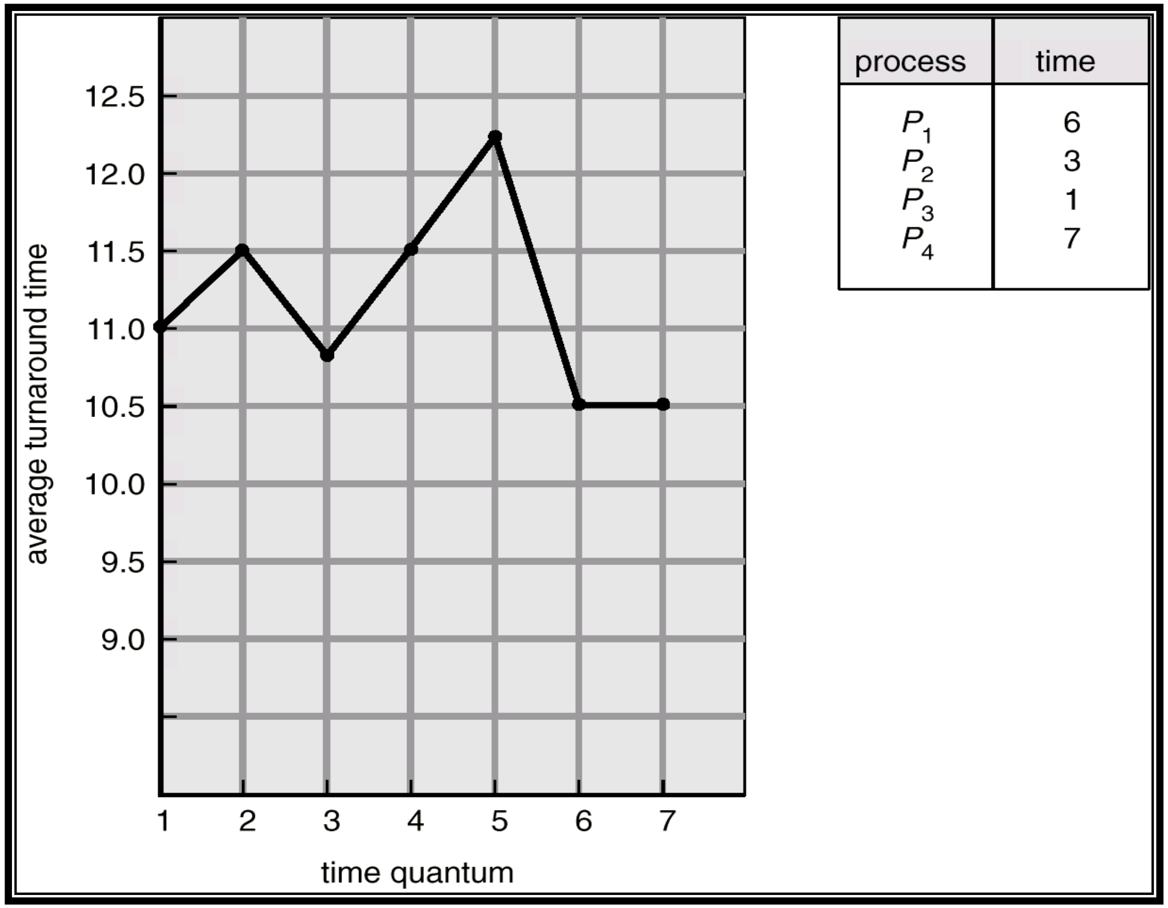
TQ: 2

Process	Arrival Time	Burst Time	Completion Time
P1	0	4	
P2	1	5	
P3	2	2	
P4	3	1	
P5	4	6	
P6	6	3	

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

5 3 1 5 1 2

15+
8+
9+
17 = 49

49/4 = 12.25

Multilevel Queue

A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues.

For Example: a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student/ user processes

Multilevel Queue

- A process can move between various queues
- Multilevel Queue Scheduler defined by the following parameters:
  - No. of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade / demote a process
  - Method used to determine which queue a process will enter and when that process needs service.

Multilevel Queue

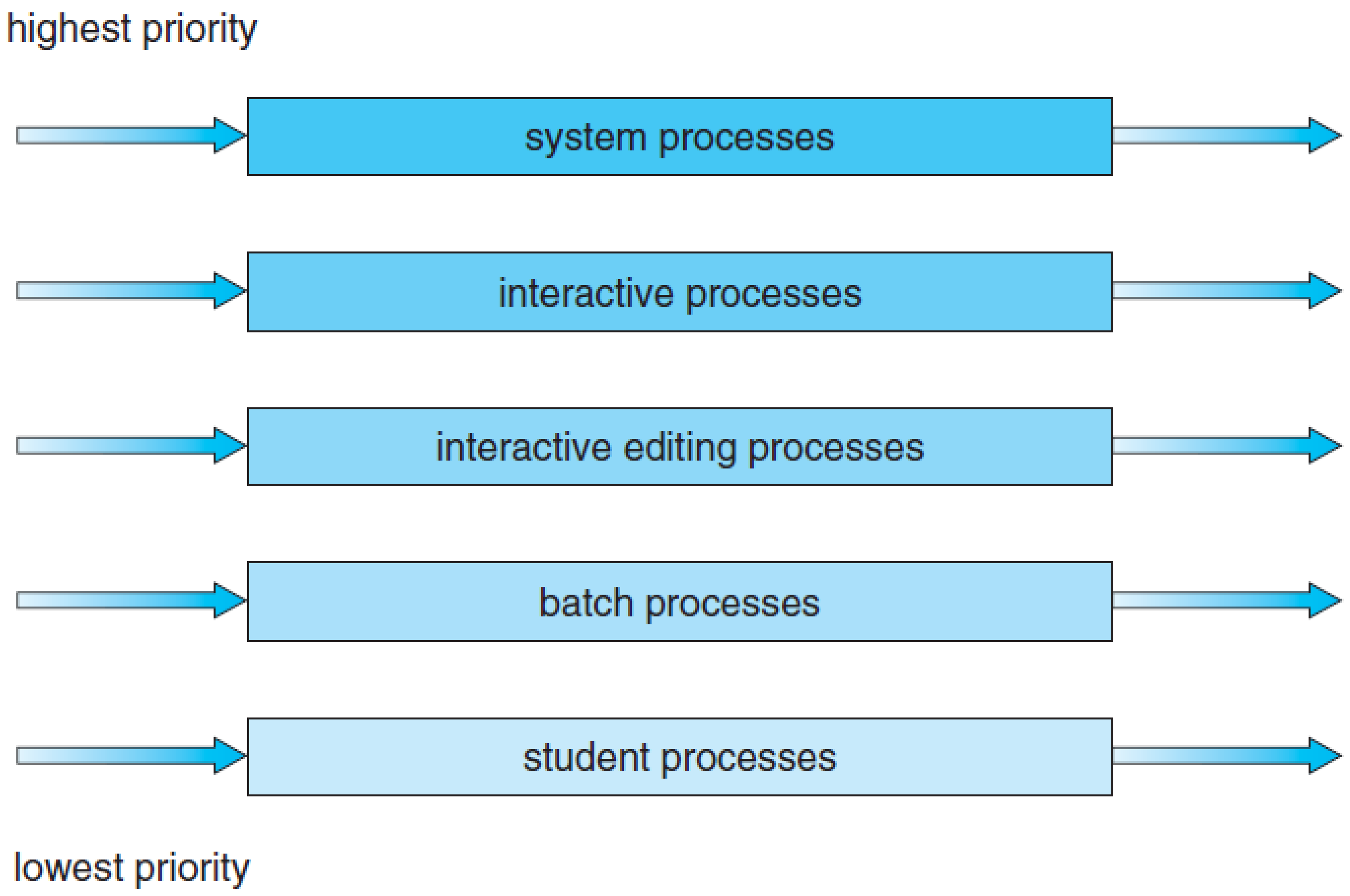
Example:

- No process in the batch queue, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.
- If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Multilevel Queue

- Processes can be :
  - Foreground Process: processes that are running currently → **RR Scheduling is applied**
  - Background Process: Processes that are running in the background but its effects are not visible to user. →**FCFS**
- Multilevel queue scheduling divides ready queue into several queues.
- Processes are permanently assigned to one queue on some property like memory size, process priority, process type.
- Each queue has its own scheduling algorithm

Multilevel Queue



Multilevel Queue

- As different type of processes are there so all cant be put into same queue and apply same scheduling algorithm.

Disadvantages:

1. **Until high priority queue is not empty**, No process from lower priority queues will be selected.
2. Starvation for lower priority processes

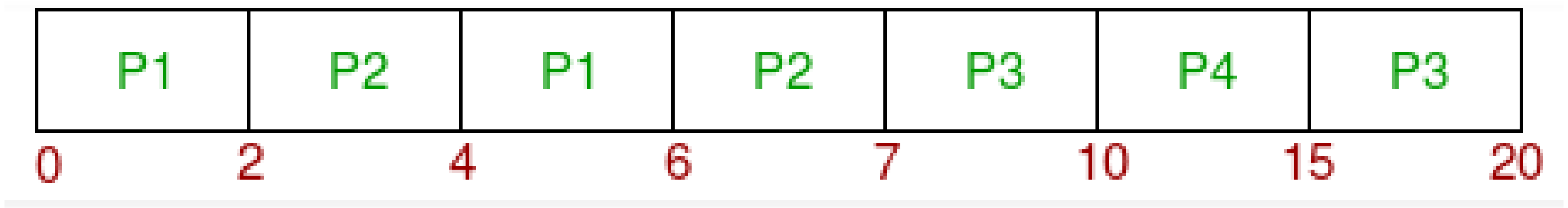
Advantage:

Can apply separate scheduling algorithm for each queue.

Practice: Multilevel Queue

Process	Arrival Time	Burst Time	Queue
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

Priority of queue 1 is greater than queue 2. queue 1 uses Round Robin (Time Quantum = 2) and queue 2 uses FCFS.


P1
P2
P1
P2
P3
P4
P3
0
2
4
6
7
10
15
20

Multilevel Feedback Queue

- Solution is: Multilevel Feedback Queue
- If a process is taking too long to execute.. Pre-empt it send it to low priority queue.
- Don’t allow a low priority process to wait for long.
- After some time move a least priority process to high priority queue → **Aging**

Multilevel Feedback Queue

✓Allows a process to move between queues.

✓The idea is to separate processes according to the characteristics of their CPU bursts.

✓If a process uses too much CPU time, it will be moved to a lower-priority queue.
✓This scheme leaves I/O-bound and interactive processes in the higher-priority queues.

✓In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

Multilevel Feedback Queue

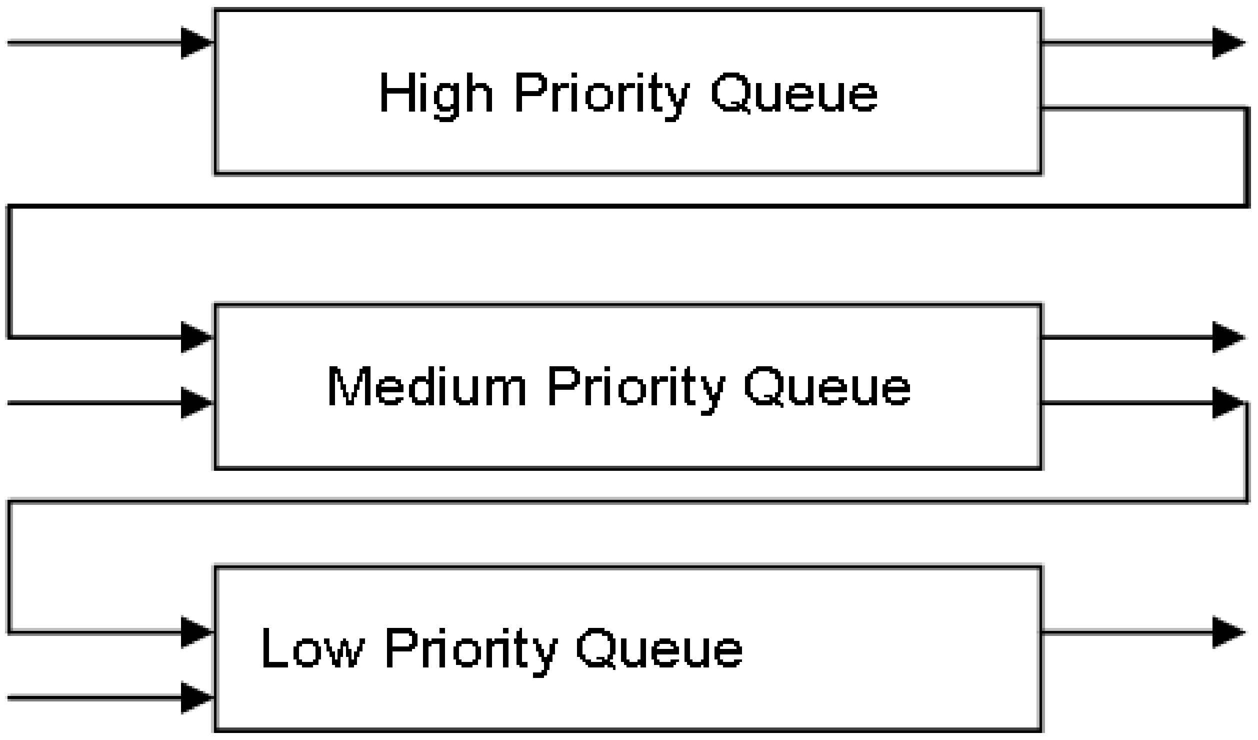
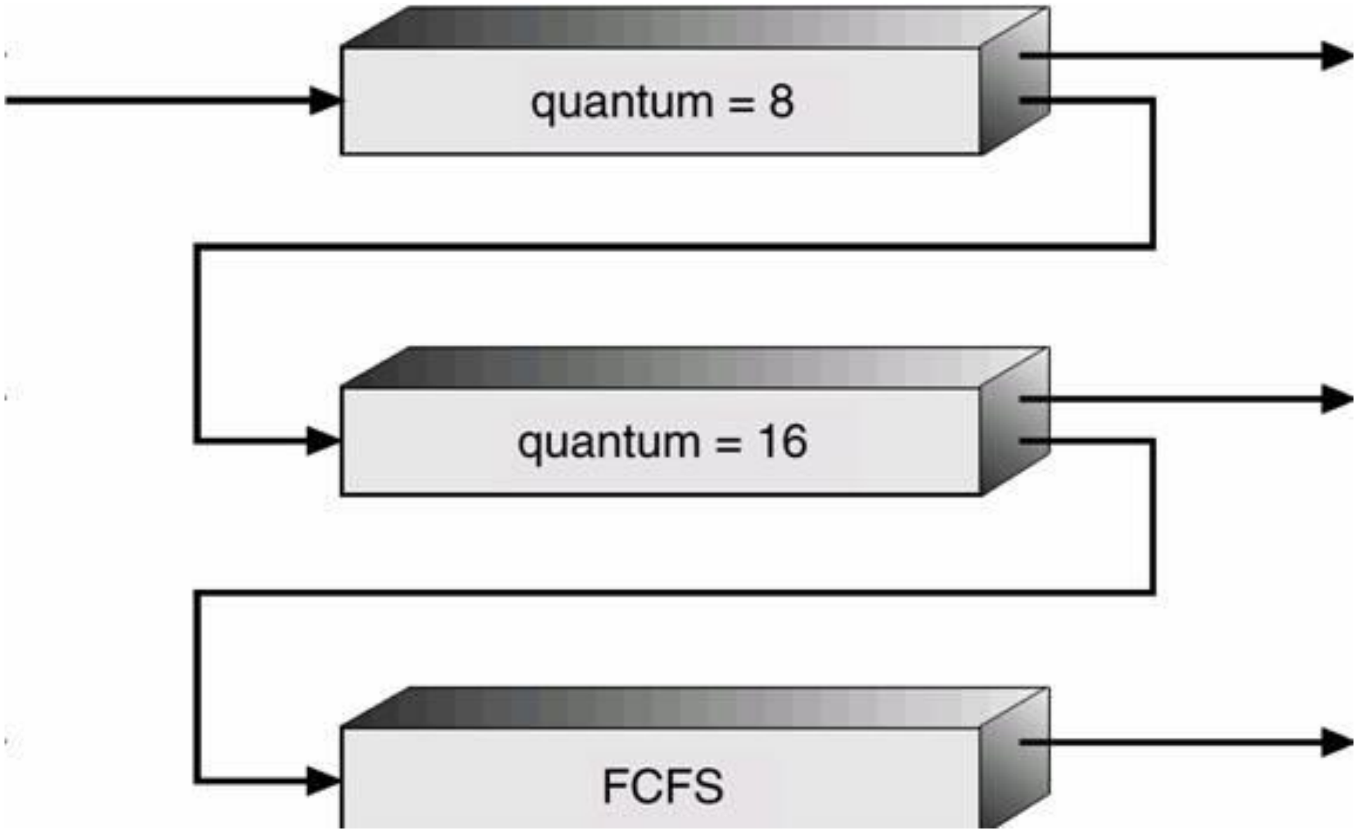


Figure – Multilevel Feedback Queue Scheduling

Multilevel Feedback Queue



Multi-processor Scheduling

Concerns:

- If multiple CPUs are available, **load sharing** becomes possible.

  - Concentration is on systems in which the **processors are identical—homogeneous in terms of their functionality**.

  - Use any available processor to run any process in the queue

Approaches to Multiple-Processor Scheduling

- **1. Asymmetric multiprocessing**

All scheduling decisions, I/O processing, and other system activities **handled by a single processor—the master server**. The other processors execute only user code.

- Only one processor accesses the system data structures, reducing the need for data sharing.

Approaches to Multiple-Processor Scheduling

 2. Symmetric multiprocessing (SMP)

•Each processor is self-scheduling.

•All processes may be in a **common ready queue**, or each processor may have its **own private queue** of ready processes.

•Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.

Virtually all modern operating systems support SMP, including Windows, Linux, and Mac OS X

Issues concerning SMP systems

- **1. Processor Affinity**

(a process has an affinity for the processor on which it is currently running.)

- Consider what happens to cache memory when a process has been running on a specific processor?

The data most recently accessed by the process populate the cache for the processor. As a result, successive memory accesses by the process are often satisfied in cache memory.

Issues concerning SMP systems

1. Processor Affinity

- If the process migrates to another processor.
- The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated.

Issues concerning SMP systems

- **1. Processor Affinity**

Because of the **high cost of invalidating and repopulating caches**, most SMP systems try to **avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor**.

This is known as **processor affinity—that is, a process has an affinity for the processor on which it is currently running.**

Issues concerning SMP systems

Forms of Processor Affinity

1. **Soft affinity**

When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so—we have a situation known as soft affinity.

2. Hard affinity

Assign a processor to a particular process and do not allow it to migrate. Guarantees that process will remain on a single processor.

Issues concerning SMP systems

Hard affirnity

Example:

Many systems provide both soft and hard affinity.

For example, Linux implements soft affinity, but it also provides the **schedsetaffinity()** system call, which supports hard affinity.

Issues concerning SMP systems

2. Load Balancing

On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.

Need of Load Balancing

One or more processors may sit idle while other processors have high workloads, along with lists of processes awaiting the CPU.

Issues concerning SMP systems

2. Load Balancing

Load balancing is necessary only on systems **where each processor has its own private queue** of eligible processes to execute.

On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue.

In most operating systems that support SMP, each processor have a private queue of eligible processes.

Issues concerning SMP systems

2. Load Balancing

Two approaches to load balancing: **push migration and pull migration.**

1. Push migration: a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by **moving (or pushing) processes from overloaded to idle or less-busy processors**.

2. Pull migration: occurs when an idle processor **pulls a waiting task from a busy processor.**

Issues concerning SMP systems

3. Multicore Processors

- A **multi-core processor** is a single computing component with two or more independent processing units called cores, which read and execute program instructions.

- A processor, or more commonly a CPU, is an individual processing device. It may contain multiple cores.

- **A core is a bank of registers and dedicated cache**

- Core is a structure that performs all of a processor's tasks, but is not an entire processor.

Issues concerning SMP systems

3. Multicore Processors

The instructions are ordinary CPU instructions (such as add, move data, and branch) but the single processor can run multiple instructions on separate cores at the same time, increasing overall speed for programs.

Manufacturers typically integrate the cores onto a single integrated circuit

Issues concerning SMP systems

3. Multicore Processors

SMP systems have allowed **several threads to run concurrently by providing multiple physical processors**.

In computer system, **multiple processor cores are placed on the same physical chip**, resulting in a **multicore processor.**

Each core maintains its architectural state and appears to the operating system to be a separate physical processor.

Issues concerning SMP systems

3. Multicore Processors

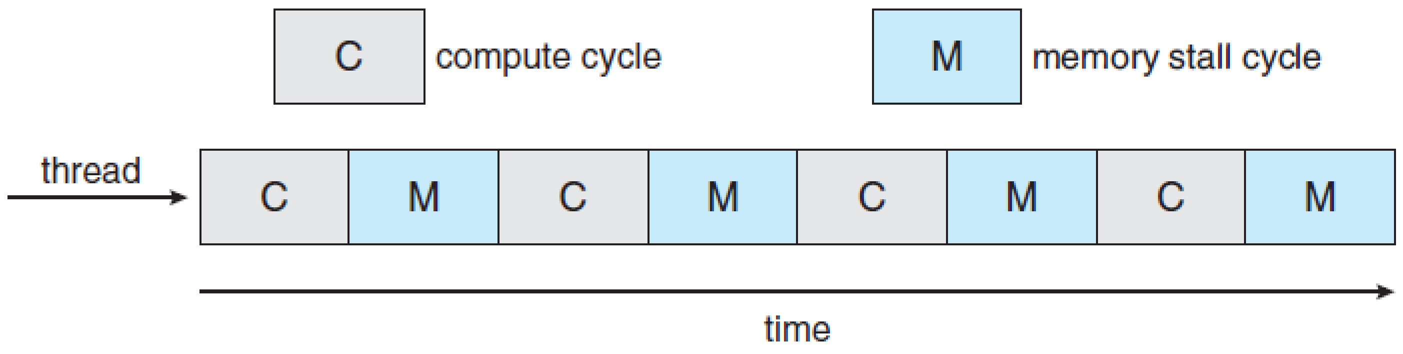
Multicore processors may complicate scheduling issues:

When a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a **memory stall.**

Memory Stall may occur due to a cache miss (accessing data that are not in cache memory).

Issues concerning SMP systems

Memory Stall



the processor can spend up to 50 percent of its time waiting for data to become available from memory.

Issues concerning SMP systems

Solution to Memory Stall Problem

To remedy this situation, hardware designs have implemented multithreaded processor cores in which **two (or more) hardware threads are assigned to each core**.

If one thread stalls while waiting for memory, the core can switch to another thread.

Real Time Scheduling

A real-time operating system (RTOS) is intended to serve real-time applications that process data without buffer delays.

a) soft real-time systems b) hard real-time systems

a) **A hard real-time system** considers timelines as a deadline, and it should not be omitted in any circumstances.

b) **A soft real-time system** is a system whose operation is degraded if results are not produced according to the specified timing requirement.

(The meeting of deadline is not compulsory for every task)

Priority based real-time scheduling

- priority-based scheduling algorithms assign each process a priority based on its importance;

- more important tasks are assigned higher priorities than less important.

- If the scheduler also supports preemption, a process currently running on the CPU will be preempted if a higher-priority process becomes available to run.

- **A preemptive, priority-based scheduler only guarantees soft real-time functionality.**

Priority based real-time scheduling

Example

Linux, Windows, and Solaris operating systems assign real-time processes the highest scheduling priority.

For example Windows has 32 different priority levels. The highest levels—priority values 16 to 31—are reserved for real-time processes.

Priority based scheduling for hard real-time

➢Hard real-time systems must guarantee that real-time tasks will be serviced in accord with their deadline requirements.

➢**Characteristics of the processes that are to be scheduled:**

1.The processes are considered **periodic.**

i.e **They require the CPU at constant intervals** (periods).

Priority based scheduling for hard real-time

➢Once a periodic process has acquired the CPU, it has a fixed processing time *t, a deadline d by which it must be serviced by the* CPU, and a period *p.*

The relationship of the processing time, the deadline, and the period can be expressed as $0 \le t \le d \le p$.

The **rate of a periodic task is 1/*p.***

Priority based scheduling for hard real-time

1. Schedulers assign priorities according to a process’s deadline or rate requirements.
2. A process may have to announce its deadline requirements to the scheduler.

3. Scheduler uses an **admission-control algorithm to perform following:**
    1. It either admits the process, guaranteeing that the process will complete on time.
    2. Rejects the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

Rate-Monotonic Scheduling

- The **rate-monotonic scheduling algorithm schedules periodic tasks using a** static priority policy with preemption.

- **rate-monotonic scheduling** (**RMS**) is a priority assignment algorithm used in real-time operating systems (RTOS) with a static-priority scheduling class.

- The static priorities are assigned according to the cycle duration of the job, so a **shorter cycle duration results in a higher job priority**

Rate-Monotonic Scheduling

- RMS assigns priorities to tasks on the basis of their periods.

- The highest-priority task is the one with the shortest period, the second highest-priority task is the one with the second shortest period, and so on.

- When more than one task is available for execution, the one with the shortest period is serviced first.

- Plot the priority of tasks as a function of their rate -> results a monotonically increasing function.

(Name is Rate Monotonic Scheduling)

Rate-Monotonic Scheduling

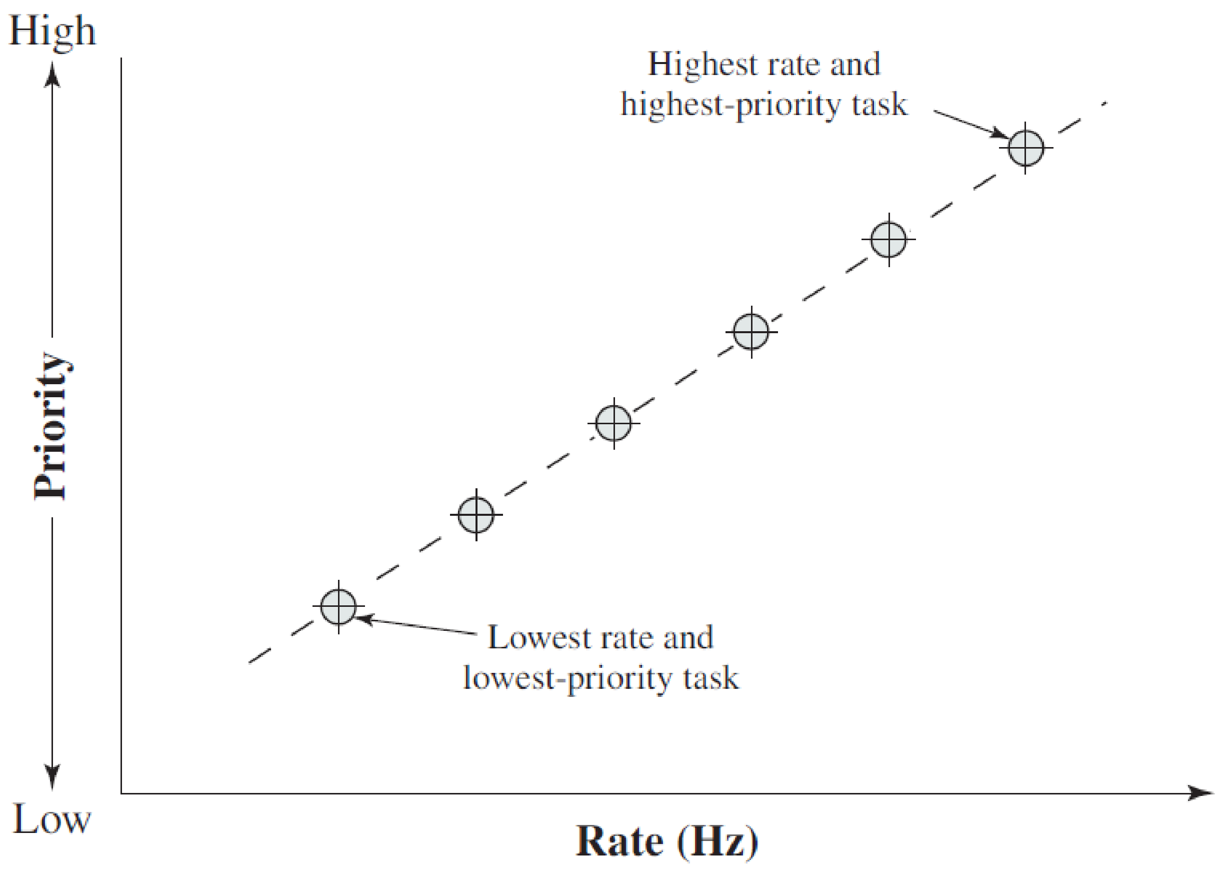


Figure 10.7 **A Task Set with RMS [WARR91]**

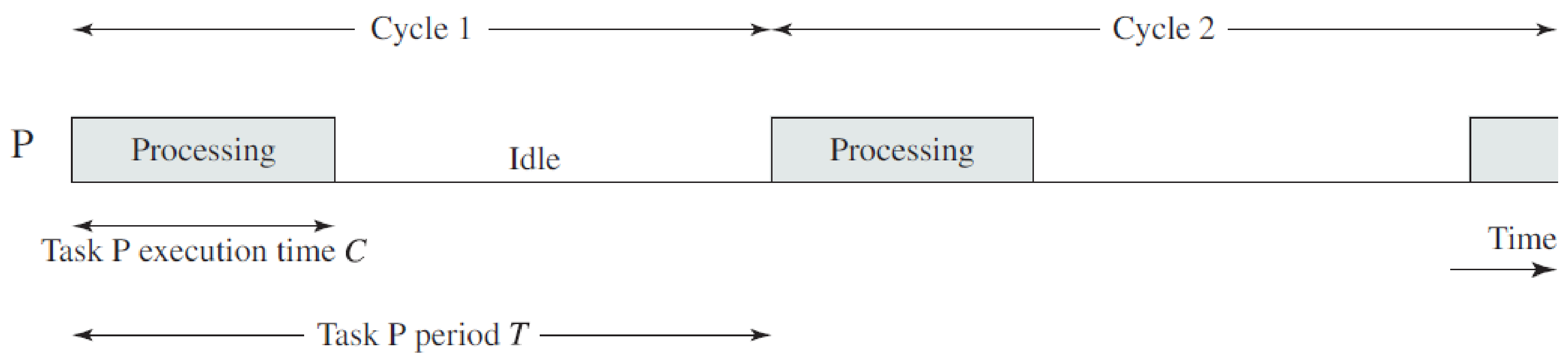


Figure 10.8 Periodic Task Timing Diagram

- The task’s period, T, is the amount of time between the arrival of one instance of the task and the arrival of the next instance of the task.
- A task’s rate is simply the inverse of its period (in seconds).
- The execution time, C, is the amount of processing time required for each Occurrence of the task.

Rate-Monotonic Scheduling

rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst.

For example
Consider P1, P2 with time period 50,100 resp. and B.T 20,35 resp.
Cal. CPU utilization of each process and total CPU utilization.
Sol:
CPU utilization=(Burst time/Time period)= (Ti/Pi)
For P1: (20/50)=0.40 i.e 40%
For P2: (35/100)=0.35 i.e 35%
Total CPU utilization is 75%

Rate-Monotonic Scheduling

rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst.

For example

Consider t1,t2,t3 with time period 100,150,350 resp. and B.T 20,40,100 resp.   U1, U2, U3 ?

Rate-Monotonic Scheduling

Consider 2 processes P1 , P2. Time Period: 50,100 resp. Burst time: 20,35. Schedule processes using Rate monotonic scheduling

Note: if CPU utilization is less than 80% only then it can be scheduled by rate monotonic scheduling. So before scheduling find Over all CPU utilization.

Consider 2 processes P1 , P2. Time Period: 50,70 resp. Burst time: 20,35. Schedule processes using Rate monotonic scheduling

Rate-Monotonic Scheduling

Rate-monotonic scheduling has a limitation:

CPU utilization is bounded, and it is not always possible fully to maximize CPU resources.

The worst-case CPU utilization for scheduling *N processes is N(2^1/N − 1)*

Earliest Deadline First Scheduling

Earliest-deadline-first (EDF) scheduling dynamically assigns priorities according to deadline.

The earlier the deadline, the higher the priority the later the deadline, the lower the priority.

Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process.