CSE308 Operating Systems

CPU Scheduling

Dr S.Rajarajan

SoC

SASTRA

Short term scheduler

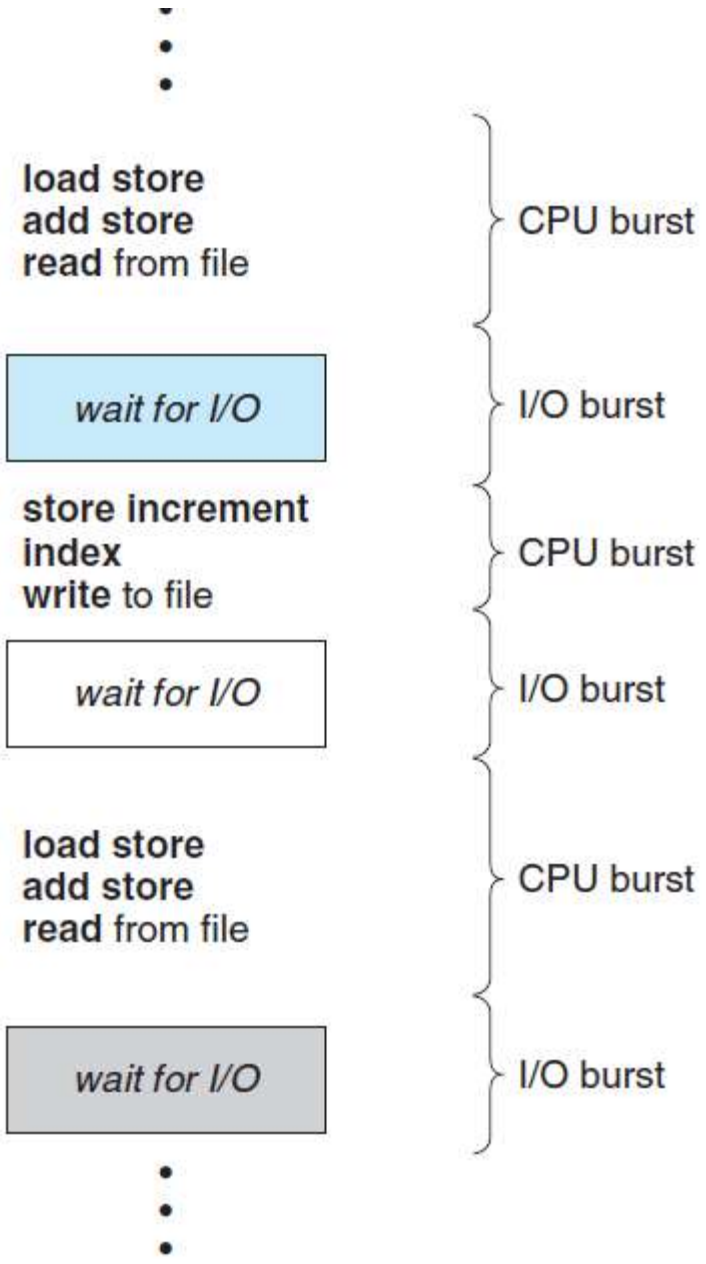
Long term scheduler

Basic Concepts

- The objective of **multiprogramming** is to have some process running at all times, to **maximize CPU utilization**.
- A process is executed until it must wait, typically for the completion of **some I/O request.**
- With multiprogramming, OS try to convert this time productively.
- By switching the CPU among processes, the operating system can make the **computer more productive** by not letting it to be **idle**
- CPU scheduling is the basis of **multiprogrammed** operating systems.

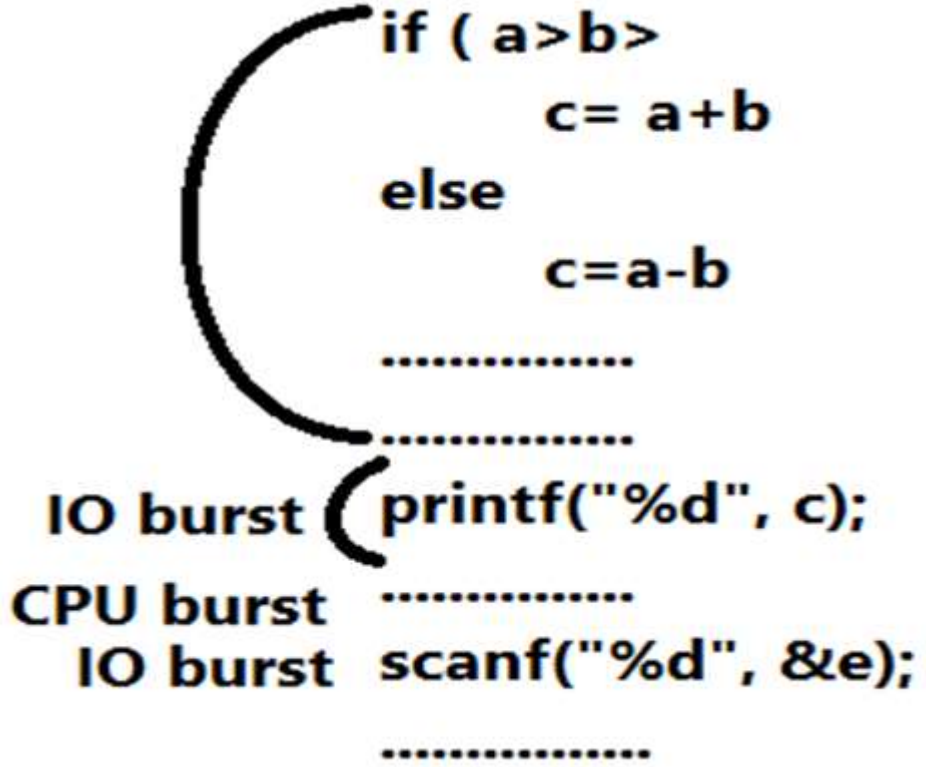
Short-term Scheduler

- **Several processes** are kept in **memory** at one time
- When one process has to wait, the operating system takes the **CPU away from that process** and gives the CPU to another process.
- This process continues.
- Every time one process has to wait, another process can take over use of the CPU.
- Scheduling is central to operating-system design.
- **Short-term scheduler or dispatcher** does the CPU scheduling


load store
add store
read from file
CPU burst
wait for I/O
I/O burst
store increment
index
write to file
CPU burst
wait for I/O
I/O burst
load store
add store
read from file
CPU burst
wait for I/O
I/O burst

CPU–I/O Burst Cycle

- Process execution consists of a **cycle of CPU execution and I/O wait.**
- Processes alternate between these two states.
- Process execution begins with a **CPU burst.**
- That is followed by an **I/O burst, which is followed by another CPU burst, then** another I/O burst, and so on.
- Eventually, the **final CPU burst ends** with a system request to **terminate execution.**
- The **durations of CPU bursts** have been measured extensively.
- They **vary greatly from process to process** and from computer to computer
- A large number of **short CPU bursts** and a small number of **long CPU bursts.**


if (a>b>
c= a+b
else
c=a-b
.....
.....
IO burst
printf("%d", c);
CPU burst
.....
IO burst
scanf("%d", &e);
.....

- An **I/O-bound program** typically has **many short CPU bursts**.
- A **CPU-bound program** might have a **few long CPU bursts**.
- This **distribution** can be **important** in the selection of an appropriate CPU-scheduling algorithm

Scheduling Objectives

- The scheduling function should
  - Share time **fairly** among processes
  - Prevent ***starvation*** of processes
  - Use the **processor efficiently**
  - Have l**ow overhead** for implementation
  - **Prioritize processes** when necessary (e.g. real time deadlines)

Preemptive Scheduling

- CPU-scheduling may take place under the following four circumstances:
  - When a process switches from the running state to the **waiting state** (for example, as the **result of an I/O request** or an invocation of **wait()** for the termination of a child process)
  - When a process switches from the **running state** to the **ready state** (for example, when an **interrupt occurs**).
  - When a **high priority** suddenly emerges and it needs to be executed immediately
  - When the **time quantum** assigned to the process is over
  - When a **process terminates**

- When scheduling takes place only under circumstances **1 and 5**, we say that the scheduling scheme is **non-preemptive or cooperative.**
- Otherwise, it is **preemptive.**
- Under **non-preemptive scheduling,** once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by **terminating** or by **voluntarily switching** to the waiting state.
- Unfortunately, preemptive scheduling can result in **race conditions** when data are shared among several processes.

- Preemption also affects **the design of the operating-system kernel.**
- During the **processing of a system call** for an **I/O**, the kernel maybe busy with that activity on the **request of a process**
- What happens **if the process is preempted** in the **middle of these changes** and the kernel (or the device driver) needs to read or modify the same structure? **Chaos** ensues.
- Certain operating systems, including most versions of UNIX, deal with this problem by **waiting for a system call to complete** before doing a context switch.
- This scheme ensures that the kernel structure is simple, since the **kernel will not preempt** a process while the kernel data structures are in an **inconsistent state**.

Dispatcher

- Another component involved in the CPU-scheduling function is the **dispatcher.**
- The dispatcher is the module that **gives control of the CPU to the process selected** by the **short-term scheduler**.
- This function involves the following:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program (with the **help of PC**)

- The dispatcher **should be as fast as** possible, since it is invoked during every process switch
- The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency** and it has to be minimal.

Scheduling Criteria

- Different CPU-scheduling algorithms have different properties, and the **choice of a particular algorithm** may favor one class of processes over another.
- In choosing which algorithm to use in a particular situation, we must consider the **properties of the various algorithms**.
  - CPU utilization
  - Throughput
  - Turnaround time
  - Waiting time
  - Response time

- **CPU utilization.** We want to keep the **CPU as busy** as possible. Conceptually, CPU utilization can range **from 0 to 100 percent**. In a real system, it should range from **40 percent to 90 percent**
- **Throughput.** One measure of work is the **number of processes** that are **completed per time unit**, called throughput**.**
- **Turnaround time (TAT).** From the point of view of a particular process, the important criterion is **how long it takes** to execute that process.
- The **interval** from the time of **submission of a process to** the **time of completion** is the turnaround time **(CT – AT)**
- **Waiting time(WT)** Waiting time is the sum of the periods spent waiting in the **ready queue (TAT-BT)**

- **Response time (RT)** In an interactive system, turnaround time may not be the best criterion.
- Often, a process can **produce some output fairly early** and can continue computing new results while previous results are being output to the user.
- Thus, **response time** is the **time from the submission of a request** until the **first response** is produced.
- RT = AT- Execution commenced time

- It is desirable to
  - **maximize CPU utilization** and **throughput**
  - **minimize turnaround time**, **waiting time**, and **response time**.
- In most cases, we **optimize the average measure**.
- However, under some circumstances, we prefer to optimize the **minimum or maximum values** rather than the average.
- It is more important to **minimize the variance** in the response time than to **minimize the average response time.**

Starvation

- When there are n processes waiting in the queue for CPU, if any process or processes **wait in the queue forever** without getting a chance then they are said to be **starving**
- Algorithms that prioritize processes based on some criteria are expected to cause starvation

CPU burst time

- We discuss various CPU-scheduling algorithms in the following section.
- An accurate illustration should involve processes each with a sequence of several hundred **CPU bursts and I/O bursts.**
- But for simplicity we consider **only one CPU burst** (in milliseconds) per process in our initial examples.

Scheduling Algorithms

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be **allocated the CPU**.
- There are many different CPU-scheduling algorithms.
  - FCFS
  - SJF
  - Priority
  - RR
  - Multilevel queue scheduling
  - Multilevel feedback queue scheduling

First-Come, First-Served 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.**
- When a process enters the **ready queue**, its **PCB** is linked onto the **tail of the queue**.
- When the **CPU is free**, it is allocated to the process at the **head of the queue**.
- The code for FCFS scheduling is **simple to write and understand**.

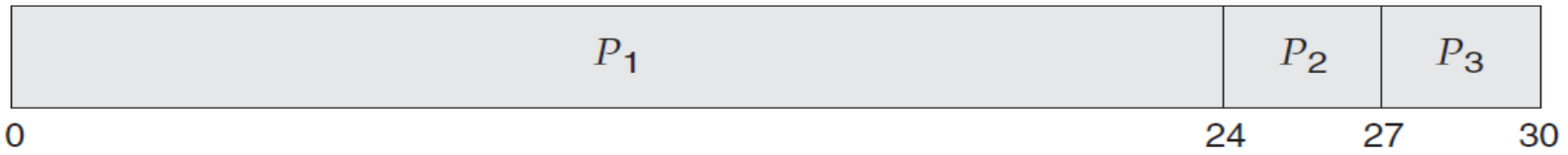
- Note also that the FCFS scheduling algorithm is **non-preemptive**.
- Once the CPU has been allocated to a process, that process **keeps the CPU until it releases** the CPU, either by **terminating** or by requesting **I/O**.

Drawback

- Poor waiting time
  - The average waiting time under the FCFS policy is often quite long
- Consider the following set of processes that arrive at time 0,
- With the length of the CPU burst given in milliseconds:

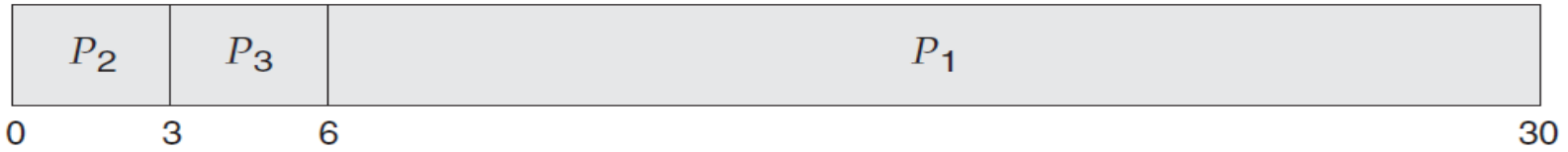
Process	Burst Time
P_1	24
P_2	3
P_3	3

- We get the result shown in the following **Gantt chart, which is a bar chart that** illustrates a particular schedule, including the start and finish times of each of the participating processes:



- The waiting time is 0 milliseconds for process *P1, 24 milliseconds for process P2, and 27 milliseconds for process P3.*
- *Thus, the average waiting time is* ***(0 + 24 + 27)/3 = 17*** milli-seconds

- If the processes arrive in the order ***P2, P3, P1,*** however, the results will be as shown in the following Gantt chart:



- The average waiting time is now **(6 + 0 + 3)/3 = 3** milliseconds.
- This reduction is substantial.
- Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

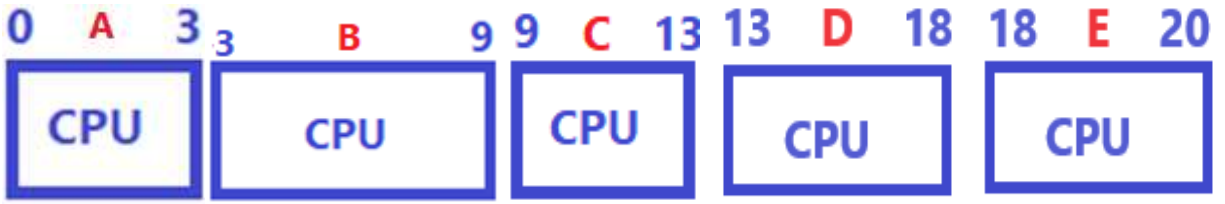
- **I/O bound processes suffer**
  - When a CPU bound process enters CPU, it holds the CPU for longer period. When IO bound process enters CPU, shortly it leaves CPU due to IO operation and joins ready queue behind CPU bound process.
- **Convoy effect**
  - As all the other processes wait for the **one big process to get off the CPU.**
  - This effect results in **lower CPU and device utilization** than might be possible if the shorter processes were allowed to go first.

Process Scheduling Example

Process	Arrival Time	Burst Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

FCFS

Process	A	B	C	D	E	
Arrival Time(AT)	0	2	4	6	8	
Service Time(BT)	3	6	4	5	2	
FCFS						
Finish Time	3	9	13	18	20	Mean
Turnaround Time (FT-AT)	3	7	9	12	12	8.60
Normalized TT (TT / BT)	1.00	1.17	2.25	2.4	6	2.56
Waiting Time (TT-BT)	0	1	5	7	10	4.6



(3+7+9+12+12)/5

Shortest-Job-First Scheduling

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

Process	A	B	C	D	E	
Arrival Time (AT)	0	2	4	6	8	
Service Time (BT)	3	6	4	5	2	
SJF						
Finish Time	3	9	15	20	11	Mean
Turnaround Time (FT-AT)	3	7	11	14	2	7.4
Normalized TT (TT / BT)	1.00	1.17	2.75	2.8	1	1.744
Waiting Time (TT-BT)	0	1	7	9	0	3.4

SJF provides better performance over FCFS for the given sample data

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

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

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

Process	Waiting time
P1	3
P2	16
P3	9
P4	0
Average WT	7

- By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

- **Merits**
  - The SJF scheduling algorithm is provably optimal, it gives the **minimum average waiting time** for a given set of processes.
  - Moving a short process before a long one **decreases the waiting time of the short proces**s more than it increases the waiting time of the long process.
  - Consequently, the average waiting time decreases
  - Throughput of CPU increases
- **Demerits**
  - Not always possible to know the burst time in advance

Predicting CPU burst time

- There is **no way to know the length** of the next CPU burst.
- One approach to this problem is to try to **approximate SJF scheduling.**
- We may not know the length of the next CPU burst, but we may be able to **predict its value**.
- We expect that the next CPU burst will be **similar in length to the previous ones**.
- By computing an **approximation of the length** of the next CPU burs t, we can pick the process with the shortest predicted CPU burst.

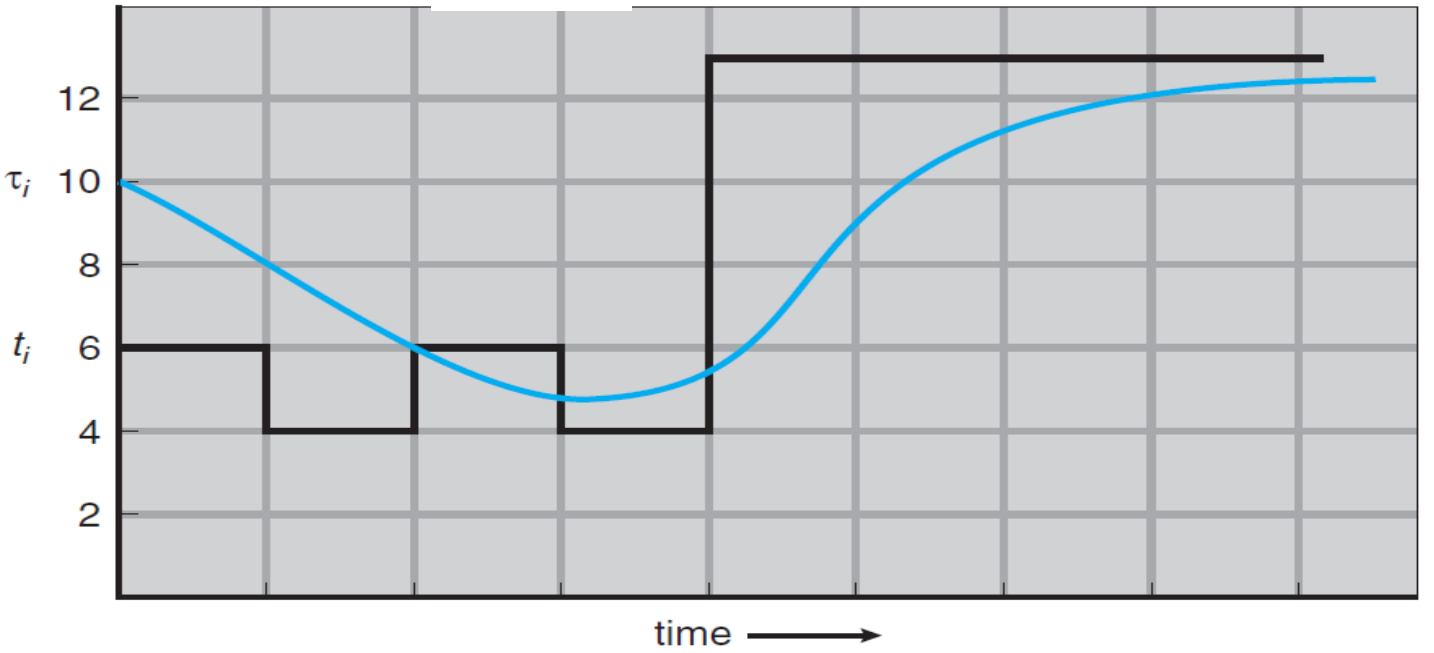
- The next CPU burst is generally predicted as an **exponential average of** the measured lengths of **previous CPU bursts.**
- We can define the exponential average with the following formula.
- Let **t_n be the length of the n^{th} CPU burst**, and let **t_{n+1} be our predicted value** for the next CPU burst.

  - **$T_{n+1} = \alpha\, t_n + (1 - \alpha)\, T_n$**

    - t_n contains our most recent information
    - **T_n** stores the past history

- The **parameter α controls the relative weight** of recent and past history in our prediction.
- **1). If α = 0, then $T_{n+1} = T_n$,** and recent history (t_n) has no effect & **past history** decides(current conditions are assumed to be transient).
- **2). If α =1, then $T_{n+1} = t_n$,** *and only the most* **recent history** matters (history is assumed to be old and irrelevant).
- 3). More commonly, α = **1/2**, so recent history and past history are **equally weighted**.
- The **initial T_0** can be defined as a **constant** or as an overall system average.

α = 0.5



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

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

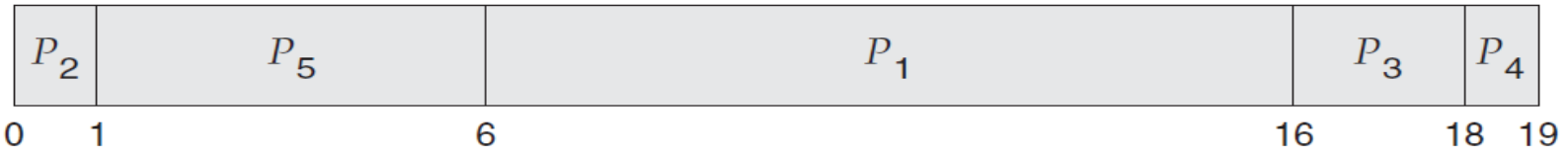
Priority Scheduling

- Apriority is associated with each process, and the CPU is allocated to the process with the highest priority.
- **Equal-priority** processes are scheduled in **FCFS** order.
- Note that we discuss scheduling in terms of ***high priority and low priority.***
- Priorities are generally indicated by some fixed range of numbers, such as **0 to 7 or 0 to 4,095**.
- However, there is no general agreement on **whether 0 is the highest or lowest priority.**
- Some systems use low numbers to represent low priority; others use low numbers for high priority.
- Non-preemptive

	P1	P2	P3	P4	P5
AT	0	0	0	0	0
BT	10	1	2	1	5
Priority	3	1	4	5	2
FT	**16**	**1**	**18**	**19**	**6**
TT	**16**	**1**	**18**	**19**	**6**

0-1	P2
1-6	P5
6-16	P1
16-18	P3
18-19	P4

The average waiting time is 8.2((6+ 0 + 16 + 18 + 1) /5)



- Priorities can be defined either **internally or externally.**
- Internally defined priorities use some measurable quantity or quantities to compute the priority of a process.
- For example,
  - time limits,
  - memory requirements,
  - the number of open files,
  - and the ratio of average I/O burst to average CPU burst
- have been used in computing priorities.

- External priorities are set by **criteria outside the operating system**, such as
  - the **importance of the process**
  - the **type and amount of funds** being paid for computer use
  - the **department sponsoring** the work
  - and other, often **political, factors**
- Priority scheduling can be either **preemptive or non-preemptive.**
- When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

- A major problem with priority scheduling algorithms is **indefinite blocking,** or **starvation.**
- A priority scheduling algorithm can leave some low priority processes waiting indefinitely.
- A solution to the problem of indefinite blockage of low-priority processes is **aging.**
- **Aging** involves gradually increasing the priority of processes that wait in the system for a long time.
- For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
- Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time	3 1	6 4	4 3 0	5 0	2 0	
Priority	6	3	2	1	2	
			Priority			
Finish Time	19	18	13	10	14	Mean
Turnaround Time	3					
Normalized TT (TT / BT)						

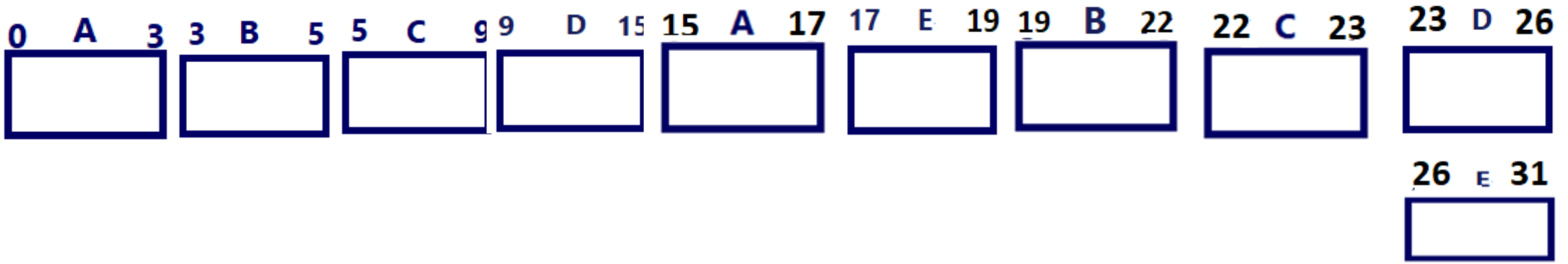
0 A 2 2 B 4 4 C 5 5 D 10 10 C 13 13 E 14 14 B 18 18 A 19

CPU Scheduling with IO burst time included (FCFS)

Process	A	B	C	D	E
AT	0	2	2	5	3
BT1	3	2	4	6	2
I/O					
BT2	2	3	1	3	5

WT = TT - (BT1+IO+BT2)

	A	B	C	D	E
CT	17	22	23	26	31
TT	17	20	20	21	24
WT	9	11	12	10	13



Preemptive Vs Non-Preemptive SJF

- The **SJF algorithm** can be either preemptive or non-preemptive.
- The choice arises when a **new process arrives** at the ready queue while a previous process is still executing.
- The next CPU burst of the **newly arrived process** may be **shorter than** what is left of the **currently executing** process.
- A preemptive SJF algorithm will **preempt the currently executing process,** whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first (SRT)** scheduling.

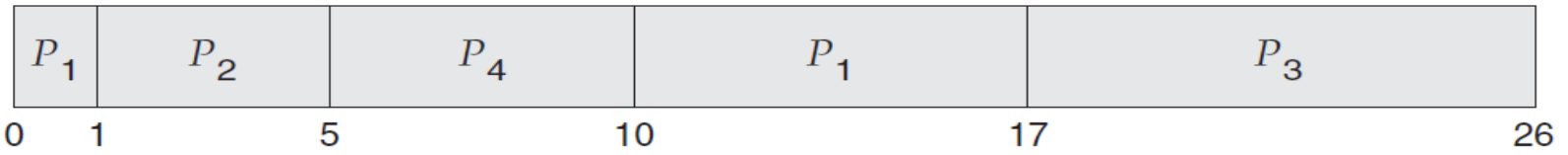
- Consider the following four processes, with the length of the CPU burst given in milliseconds:

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

Preemptive SJF or SRT

	P1	P2	P3	P4
AT	0	1	2	3
BT	8	4	9	5
FT	17	5	26	10

0-1	P1(7)
1-5	P2(0)
5-10	P4(0)
10-17	P1(0)
17-25	P3 (0)



- The average waiting time for this example is [(10 – 1) + (1 – 1) + (17 – 2) + (5 – 3)]/4 = 26/4 = **6.5 milliseconds**.
- **Non-preemptive SJF** scheduling would result in an average waiting time of **7.75 milliseconds**
- A major problem with SJF algorithms is **indefinite blocking,** or **starvation**
- A process that is ready to run but waiting for the CPU can be considered blocked.
- A SJF algorithm can leave some **large BT** processes **waiting indefinitely**.
- In a heavily loaded computer system, **a steady stream** of short processes can prevent a long process from ever getting the CPU.

Round-Robin Scheduling

- The **round-robin (RR)** scheduling algorithm is designed especially for **timesharing**
- It is **similar to FCFS** scheduling, but **preemption is added** to enable the system to switch between processes
- A small unit of time, called a **time quantum** or **time slice**, is defined
- A time quantum is generally from**10 to 100** milliseconds in length
- The ready queue is treated as a **circular queue**

- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to **1 time quantum.**
- To implement RR scheduling, we again treat the **ready queue as** a **FIFO queue** of processes.
- New processes are added to the tail of the ready queue.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum (TQ), and dispatches the process.

- One of two things will then happen.
  - BT = TQ – process will complete TQ and process leaves.
  - BT > TQ – process will complete TQ, pre-empted and joins the ready queue.
  - BT < TQ - the process itself will release the CPU voluntarily.
- The **average waiting time** under the RR policy is often long.

Find the TT for the following processes scheduled using RR with TQ=1

Process	A	B	C	D	E
AT	0	2	4	6	8
BT	1	6	4	5	2
FT	4	18	17	20	15

CPU	Queue
0-1 – A	
1-2- A	B
2-3 – B	A
3-4 – A	BC
4-5- B	C
5-6 – C	BD
6-7 – B	DC
7-8 – D	CBE

14-15 – E2	DCB
15-16 – D3	CB
16-17 – C4	BD
17-18 – B6	D
18-19 – D4	
8-9 – C	BED
9-10 – B4	EDC
10-11 - E1	DCB
11-12 – D2	CBE
12-13 - C3	BED
13-14 – B5	EDC
19-20 – D5	

Compute the finish times of processes based on RR with time quantum as 4

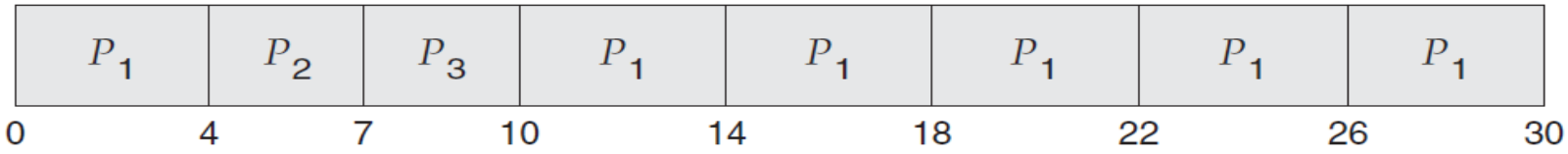
Process	A	B	C	D	E
AT	0	2	4	6	8
ST	3	6	4	5	2
FT	3	17	11	20	19

CPU	Queue
0-3 A3	
3-7- B4	CD
7-11– C4	DBE
11-15– D4	BE
15-17- B6	ED
17-19 – E2	D
19-20 – D5	

- Consider the following set of processes that **arrive at time 0**, with the length of the CPU burst given in milliseconds (TQ=4):

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Calculate the average waiting time for this schedule

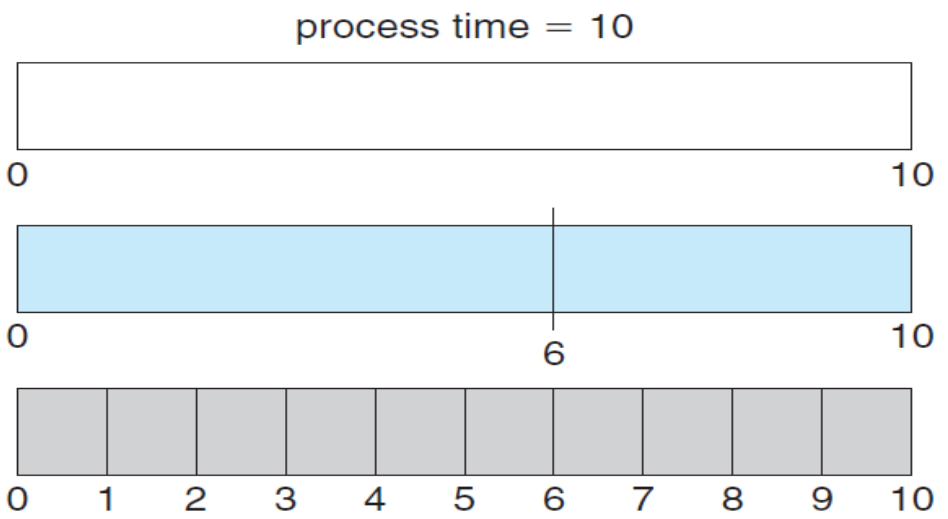


- *P1 waits for 6* milliseconds (10 - 4), *P2 waits for 4 milliseconds, and P3 waits for 7 milliseconds.*
- Thus, the average waiting time is 17/3 = 5.66 milliseconds.

- 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.*
- Each process must wait **no longer than (*n – 1) × q time units*** until *its* next time quantum.
- For example, with **5 processes** and a **time quantum of 20** milliseconds, each process will get up **to 20 milliseconds every 100 milliseconds.**
- **Performance of the RR** algorithm depends on the **size of the time quantum**.
- At one extreme, if the time quantum is **extremely large**, the RR policy is the same as the **FCFS policy**.
- In contrast, if the time quantum is **extremely small** (say, 1 millisecond), the RR approach can result in a large number of **context switches**.

Controlling Context Switches

- Thus, we want the time quantum to be **large with respect to the context switch time**.
- If the context-switch time is approximately **10 percent of the time quantum**, then about **10 percent of the CPU time** will be spent in **context switching**.
- In practice, most modern systems have time quanta ranging from **10 to 100 milliseconds**.
- The time required for a context switch is typically **less than 10 microsecond**s; thus, the context-switch time is a small fraction of the time quantum.
- Turnaround time also depends on the size of the time quantum



quantum	context switches
12	0
6	1
1	9

Multilevel Queue Scheduling

- Another class of scheduling algorithms has been created for situations in which processes are **classified into different groups**
- For example, a common division is made between **foreground (interactive) processes and background (batch) processes.**
- These two types of processes have different **response-time requirements** and so may have **different scheduling needs**.
- In addition, foreground processes may have **priority** (externally defined) over background processes.

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

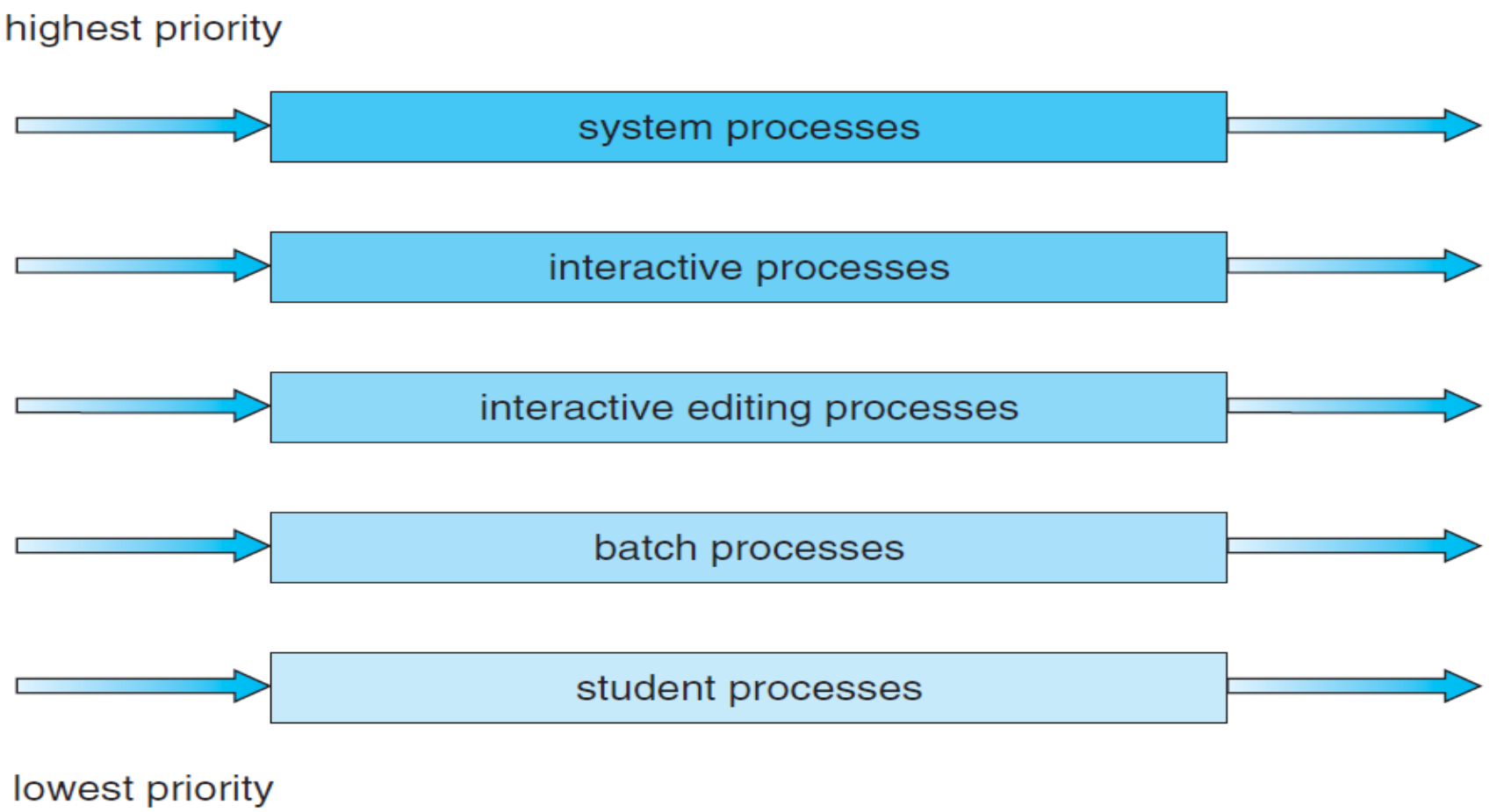


Figure 6.6 Multilevel queue scheduling.

- The **processes are permanently assigned** to one queue, generally based on some property of the process, such as **memory size, process priority, or process type.**
- Each queue has its **own scheduling algorithm**.
- The foreground queue might be scheduled by an **RR algorithm,** while the background queue is scheduled by an **FCFS algorithm**.
- In addition, there must be scheduling among the queues, which is commonly implemented as **fixed-priority preemptive** scheduling.

- Let’s look at an example of 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 processes
- Another possibility is to t**ime-slice among the queues.**
- Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.

Multilevel Feedback Queue Scheduling

- Normally, when the multilevel queue scheduling algorithm is used, processes are **permanently assigned** to a queue when they enter the system.
- **Processes do not move** from one queue to the other.
- This setup has the **advantage of low scheduling overhead**, but it is **inflexible**.
- The **multilevel feedback queue scheduling** algorithm, in contrast, 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**.

- For example, consider a multilevel feedback queue scheduler with three queues, numbered from **0 to 2**.
- The scheduler first executes all processes in queue 0.
- Only when queue 0 is empty will it execute processes in queue 1.
- Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty.
- A process that arrives for queue 1 will preempt a process in queue 2.
- A process in queue 1 will in turn be preempted by a process arriving for queue 0.

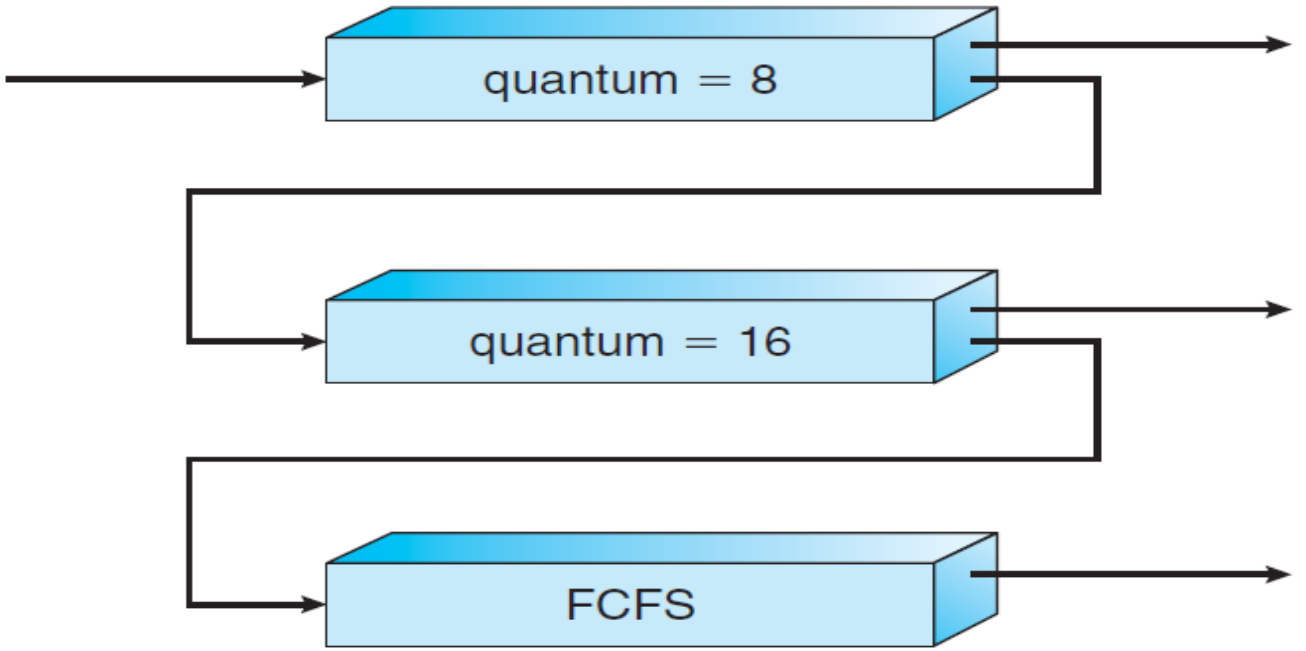
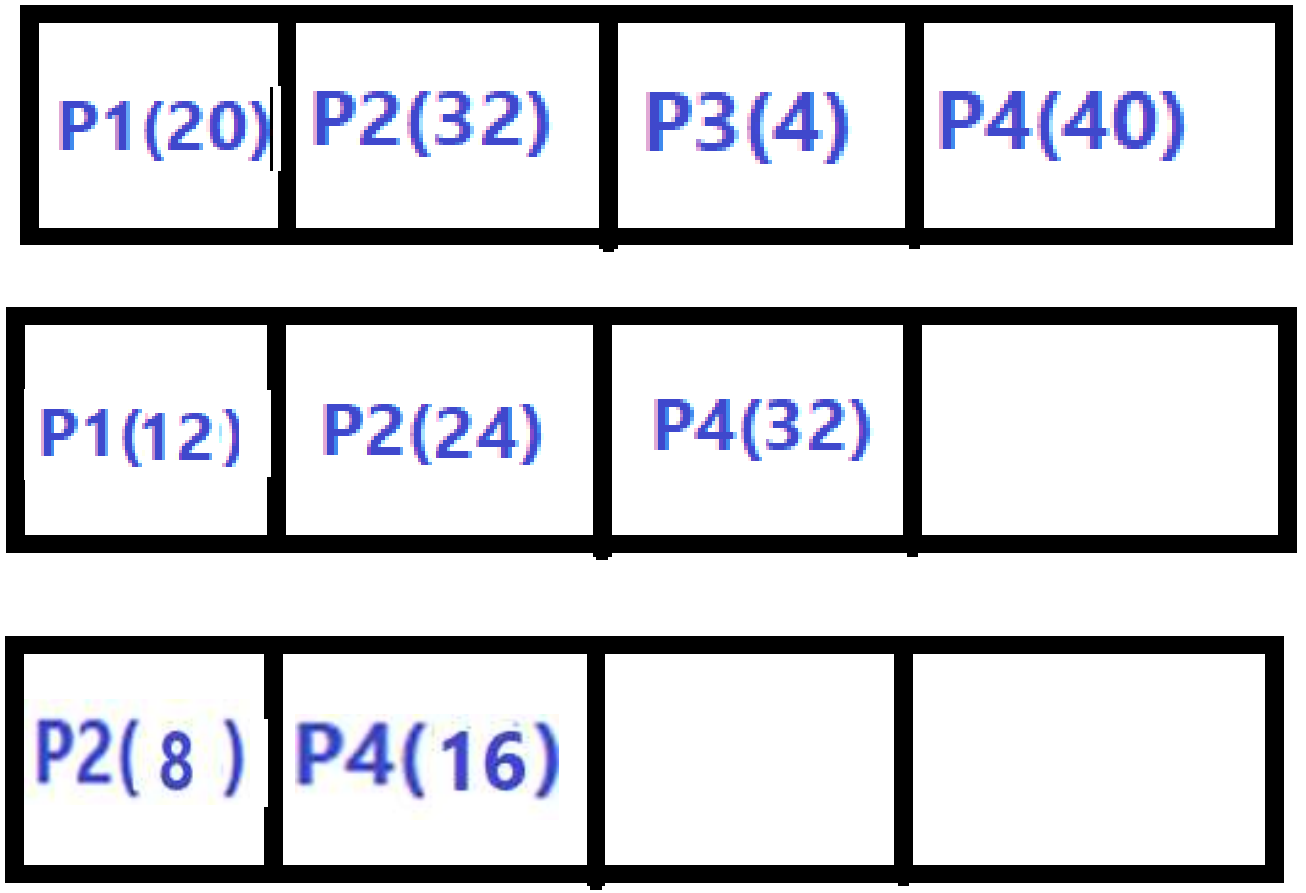


Figure 6.7 Multilevel feedback queues.


P1(20)
P2(32)
P3(4)
P4(40)
P1(12)
P2(24)
P4(32)
P2(8)
P4(16)

- In general, a multilevel feedback queue scheduler is defined by the following parameters:
  - The number of queues
  - The scheduling algorithm for each queue
  - The method used to determine when to upgrade a process to a higher priority queue
  - The method used to determine when to demote a process to a lower priority queue
  - The method used to determine which queue a process will enter when that process needs service
- The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm.
- Unfortunately, it is also the most complex algorithm

Multi-level Feedback Scheduling

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Assume that there are 3 queues with each queue following the time quantum of 2^i. Compute the mean turnaround time.

Procedure

- When a process first enters the system, it is placed in RQ0.
- After its preemption, it is placed in RQ1.
- Each subsequent time that it is preempted, it is demoted to the next lower-priority queue.
- A short process will complete quickly, without migrating very far down the hierarchy of ready queues.
- A longer process will gradually drift downward.
- Thus, newer, shorter processes are favored over older, longer processes.
- Once in the lowest-priority queue, a process cannot go lower, but is returned to this queue repeatedly until it completes execution.
- Thus, this queue is treated in **round-robin fashion.**
- **Time quantum is said to be 2^i. So RQ0 TQ= 1 (2^0), RQ1 TQ= 2(2^1), RQ2 TQ=4(2^2)**

Schedule the following processes under FBQ scheduling with 3 levels of queue and the TQ = 2^i where i is the queue level staring from 0

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

RQ0

Process	Pending Service time
A	3
B	6
C	4
D	5
E	2

RQ1(TQ=2)

Time	Process	Pending Service time
1	A(1-3)	0
	B(5-7)	3

RQ0 (TQ=1)

Time	Process	Pending Service time
0	A(0-1)	2
3	B(3-4)	5
4	C(4-5)	3
7	D(7-8)	4
8	E(8-9)	1

Note: At time 1 since no process in RQ0,
Process A waiting in RQ1 is executed for 2 TQs
At time 5 since RQ0 is empty, B from RQ1 executed
For 2 TQs

RQ1

Process	Pending Service time
B	3
C	3
D	4
E	1

RQ1(TQ=2)

Time	Process	Pending Service time
9	C(9-11)	1
11	D(11-13)	2
13	E(13-14)	0

RQ3

Process	Pending Service time
B	3
C	1
D	2

RQ3(TQ= 4)

Time	Process	Pending Service time
14	B(14-17)	
17	C(17-18)	
21	D(18-20)	

FB $q = 2^i$						
Finish Time	3	17	18	20	14	
Turnaround Time (T_r)	4	15	14	14	6	10.60
T_r/T_s	1.33	2.50	3.50	2.80	3.00	2.63

FB $q = 1$						
Finish Time	4	20	16	19	11	
Turnaround Time (T_r)	4	18	12	13	3	10.00
T_r/T_s	1.33	3.00	3.00	2.60	1.5	2.29

	FCFS	Round robin	SPN	SRT	Feedback	Priority
Selection function	max[w]	constant	min[s]	min[s − e]	max[w]	Max[p]
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Preemptive (at time quantum)	Preemptive or non-premptive
Throughput	Not emphasized	May be low if quantum is too small	High	High	Not emphasized	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Not emphasized	Provides good response time for high priority processes
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Minimum
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	May favor I/O bound processes	Penalizes low Priority processes
Starvation	No	No	Possible	Possible	Possible	Possible