Scheduling Algorithms

Prof. Shweta Dhawan Chachra

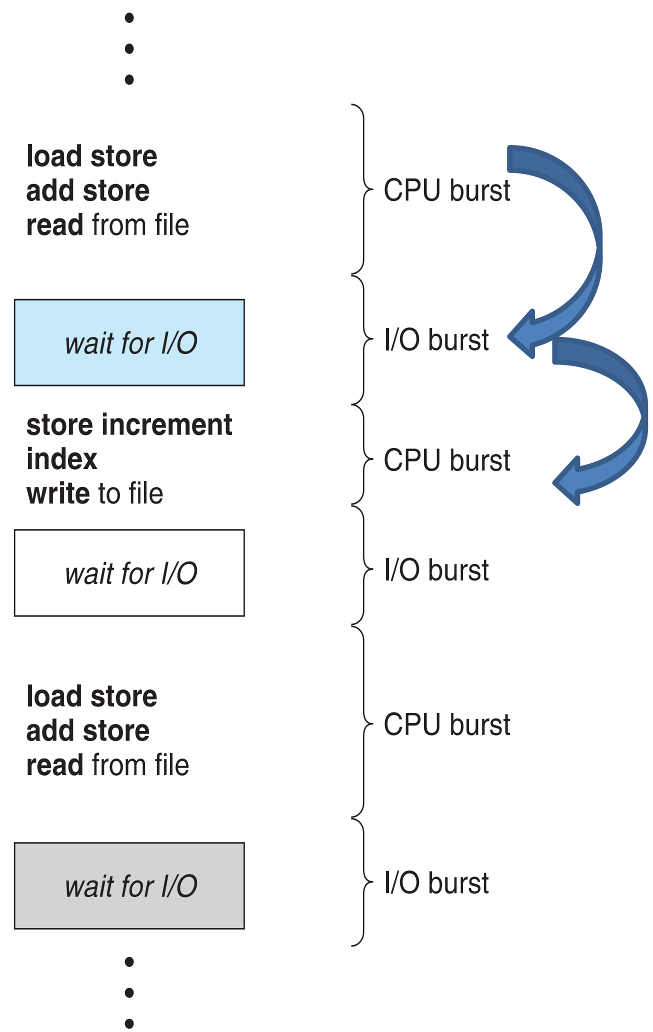
Scheduling Algorithms
Why??

- Maximum CPU utilization obtained with multiprogramming:-
  - **Switching CPU among processes to make the OS more productive**
  - Several processes kept in memory at one time.
  - When one process has to wait, the OS takes CPU away from that process and gives it to another process.

- Scheduling is fundamental OS function.

CPU–I/O Burst Cycle –

- Process execution consists of a **cycle** of CPU execution and I/O wait.
- Processes alternate between these 2 states.
  - **Process execution begins with CPU burst,**
  - **then followed by I/O burst and so on.**
  - **Last CPU burst** will end with a system request to terminate execution.
- **CPU burst distribution is of main concern**



Scheduling Algorithm Optimization Criteria

1) **CPU utilization**
2) **Throughput**
3) **Turnaround time**
4) **Waiting time**
5) **Response time**

Scheduling Criteria

- **CPU utilization** –
  - keep the CPU as busy as possible,
  - May vary from 0 to 100%.
  - In a real system, it should range from 40% (lightly loaded) to 90% (for a heavily used).

Scheduling Criteria

- **Throughput** –
  - No of processes that complete their execution per time unit

Scheduling Criteria

- **Turnaround time** –
  - amount of time to execute a particular process
  - **The interval from the time of submission of a process to the time of completion**

Scheduling Criteria

- **Waiting time** –
  – amount of time a process has been waiting in the ready queue/
  – Sum of periods spent waiting in ready queue

Scheduling Criteria

- **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)

Scheduling Algorithm Optimization Criteria

1) **CPU utilization**
2) **Throughput**
3) **Turnaround time**
4) **Waiting time**
5) **Response time**

??

Scheduling Algorithm Optimization Criteria

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

CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways

CPU Scheduler

- CPU scheduling decisions may take place when a process:
  1) **Switches from running to waiting state**

     Eg- I/O request, Invocation of wait for the termination of one of the child processes
  2) **Switches from running to ready state**

     Eg- When an interrupt occurs or timer expires
  3) **Switches from waiting to ready**

     Eg-Completion of I/O
  4) **Terminates**

CPU Scheduler

- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Non-Preemptive Scheduling

- Under non-preemptive scheduling,
  - once the CPU has been allocated to a process,
  - the process keeps the CPU
  - it releases the CPU either by terminating or by switching to the waiting state.

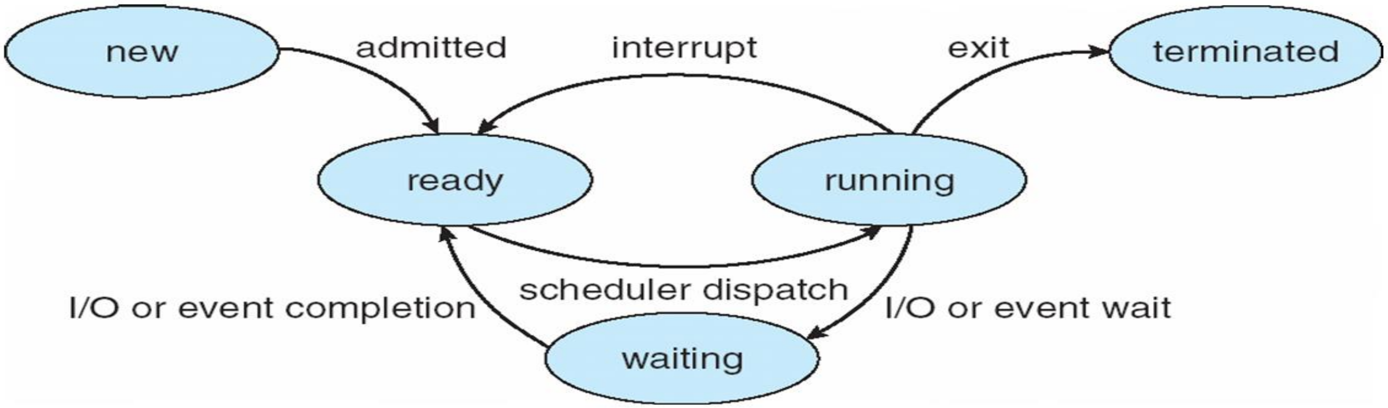
- This scheduling method was used by Microsoft Windows 3.x;

Preemptive Scheduling

- Windows 95 introduced preemptive scheduling,
- All subsequent versions of Windows operating systems have used preemptive scheduling.

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** – time it takes for the dispatcher to stop one process and start another running



Non Pre-emptive Algorithms

First- Come, First-Served (FCFS) Scheduling

- Simplest CPU scheduling Algorithm
- Process that requests CPU first, is allocated CPU first

First- Come, First-Served (FCFS) Scheduling

- Implemented using FIFO Queue
  - Process enters the ready queue, its PCB is linked onto the tail of the queue
  - When CPU is free, it is allocated to the process at the head of the queue.

- **The Average Waiting time for FCFS is often quite long.**

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

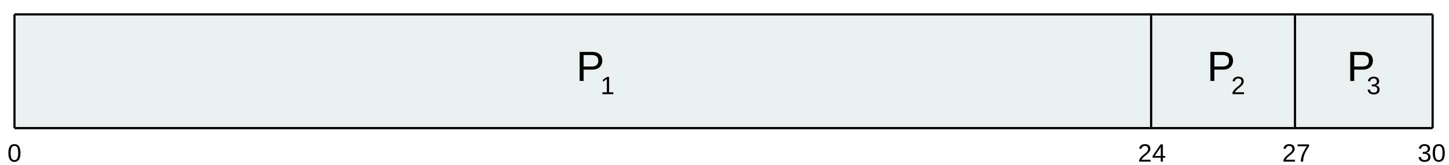
Gantt Chart

- **Generalized Activity Normalization Time Table (GANTT)**
- Horizontal bar chart developed by Henry L. Gantt (American engineer and social scientist) in 1917 as production control tool.
- Series of horizontal lines are present that show the amount of work done or production completed in given period of time in relation to amount planned for those projects.

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

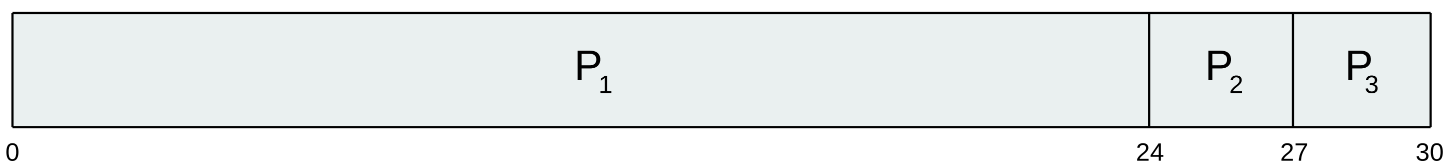
- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



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 = 51/3=17

FCFS Scheduling (Cont.)

- <u>FCFS is Non-preemptive</u>

- Once the CPU is allocated to a process, that process keeps the CPU until it releases the CPU either by terminating or by requesting I/o.

- Not good for Time sharing system

FCFS Scheduling (Cont.)

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

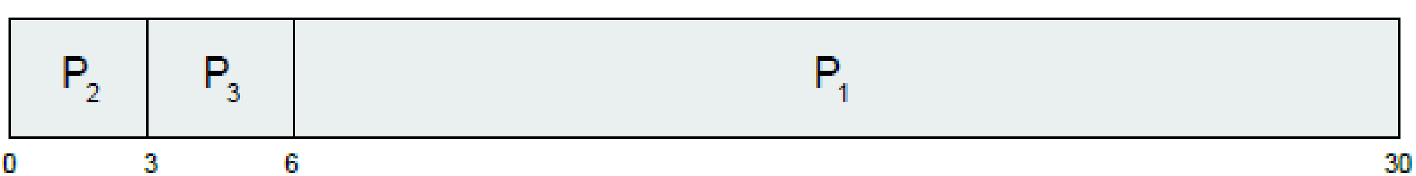
FCFS Scheduling (Cont.)

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 is:



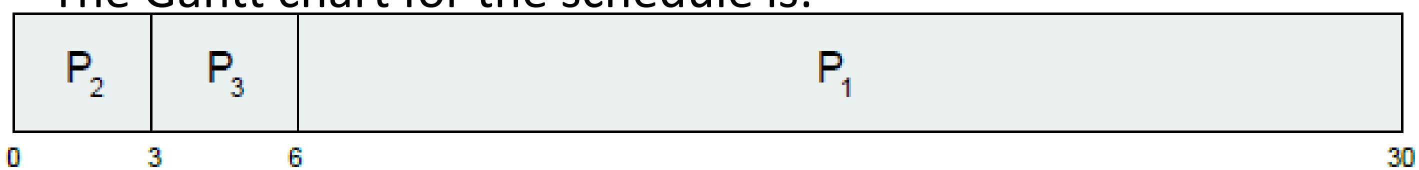
FCFS Scheduling (Cont.)

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 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

- **Convoy effect** - short process behind long process
- Phenomenon associated with the First Come First Serve (FCFS) algorithm,
- In which the whole Operating System slows down due to few large processes.

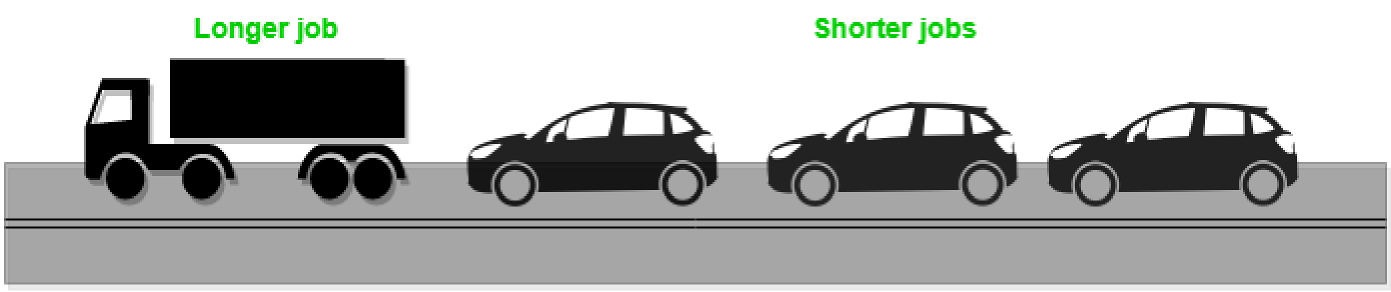


Figure - The Convey Effect, Visualized

Convoy effect

- The Convoy Effect is a phenomenon where **a large or resource-intensive process ties up system resources and causes a backlog of other processes waiting to use those same resources.**

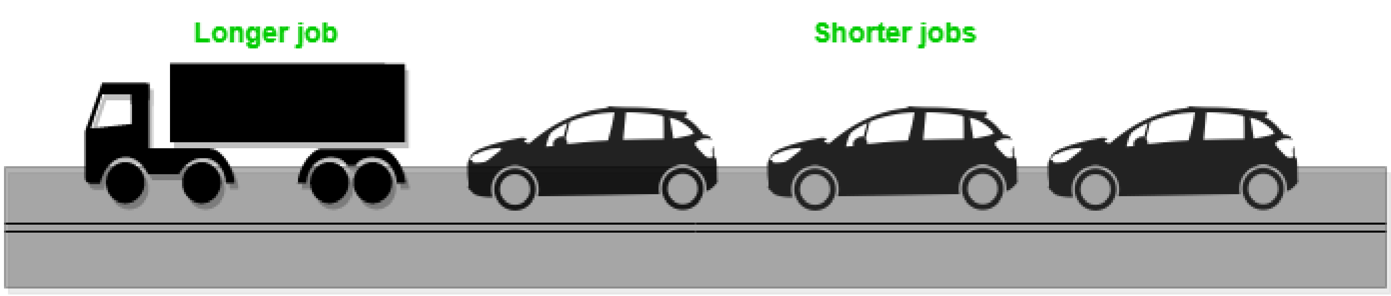


Figure - The Convey Effect, Visualized

Formulae-

1) **Turn Around Time = Completion Time - Arrival Time**
2) **Waiting Time = Turn Around Time - Burst Time or Sum of times spent waiting in Ready queue-Arrival Time**
3) **Average waiting time = Total_waiting_time / no_of_processes**
4) **Average turnaround time = Total_turn_around_time / no_of_processes**

Exercise

Consider 5 processes with process ID **P0, P1, P2, P3 and P4.** The processes and their respective Arrival and Burst time are given in the following table. Apply FCFS Scheduling algorithm and calculate the following:

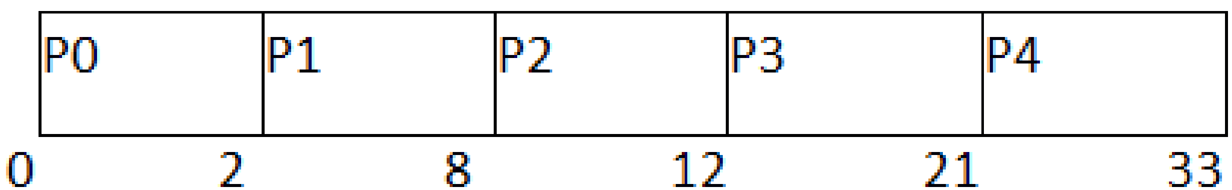
1) Turnaround time
2) Waiting time for all processes
3) Average Waiting Time

Process ID	Arrival Time	Burst Time
0	0	2
1	1	6
2	2	4
3	3	9
4	6	12

Exercise

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
0	0	2	2	2	0
1	1	6	8	7	1
2	2	4	12	10	6
3	3	9	21	18	9
4	6	12	33	27	15

Gantt Chart-



Waiting Time=Sum of time spent in Ready Queue-Arrival Time

Turnaround Time=Completion Time-Arrival Time

Exercise

Process ID	Arrival Time	Burst Time
0	0	2
1	1	6
2	2	4
3	3	9
4	6	12

WT(P0)=0-0=0
WT(P1)=2-1=1
WT(P2)=8-2=6
WT(P3)=12-3=9
WT(P4)=21-6=15

Gantt Chart-

P0	P1	P2	P3	P4
0 – 2	2 – 8	8 – 12	12 – 21	21 – 33

Exercise

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
0	0	2	2	2	0
1	1	6	8	7	1
2	2	4	12	10	6
3	3	9	21	18	9
4	6	12	33	27	15

Gantt Chart-

P0	P1	P2	P3	P4

0 2 8 12 21 33

Avg Waiting Time=31/5

Avg Turnaround Time=66/5

Shortest-Job-First (SJF) Scheduling

- An algorithm in which the process having the smallest execution time is chosen for the next execution.

- Associate with each process the length of its next CPU burst
  - **Use these lengths to schedule the process with the shortest time**

Shortest-Job-First (SJF) Scheduling

- SJF is optimal –
  - gives minimum average waiting time for a given set of processes

Shortest-Job-First (SJF) Scheduling

- Two schemes:
  - Non-preemptive
  - preemptive – Also known as the Shortest-Remaining-Time-First (SRTF).

Shortest-Job-First (SJF) Scheduling

- **Non-preemptive SJF is *optimal***
  - **if all the processes are ready simultaneously– gives minimum average waiting time for a given set of processes.**

- **SRTF is *optimal***
  - **if the processes may arrive at different times**

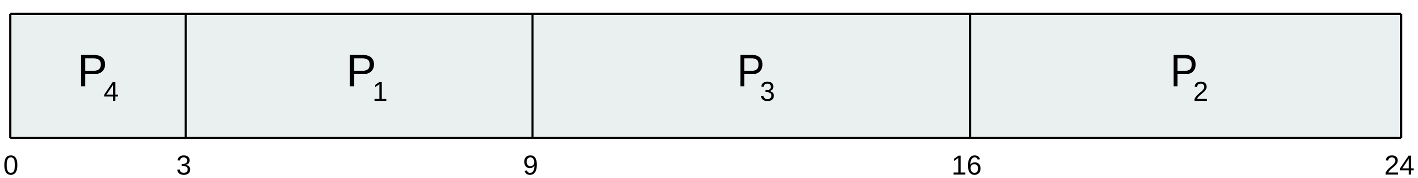
Example of Non-preemptive SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Example of Non-preemptive 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 = 28/4=7

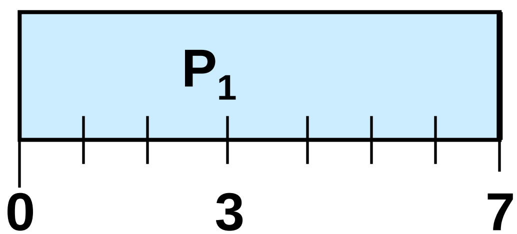
Example for Non-Preemptive SJF with different arrival times

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

Example for Non-Preemptive SJF with different arrival times

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)

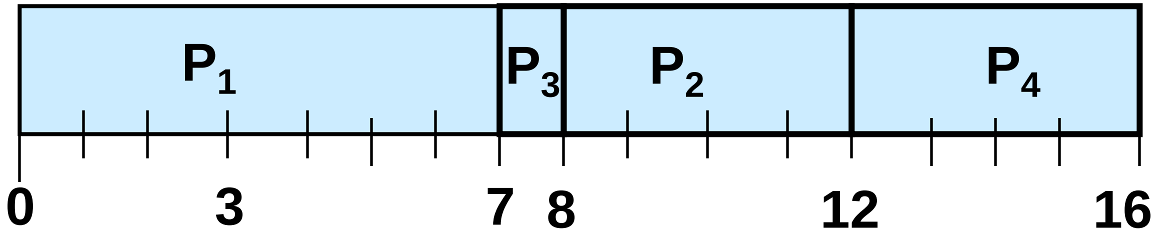


- At time 0, P_1 is the only process, so it gets the CPU and runs to completion

Example for 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

- Once P_1 has completed the queue now holds P_2, P_3 and P_4



- P_3 gets the CPU first since it is the shortest. P_2 then P_4 get the CPU in turn (based on arrival time)

Round Robin (RR)

What ?

- Designed for time sharing systems
- Similar to FCFS but preemption is added
- Time Quantum/Time Slice-small unit of time is defined, usually 10-100 milliseconds

Round Robin (RR)

What ?

- Ready queue is treated as Circular Queue

- The CPU scheduler goes around the ready queue, allocating CPU to each process for time interval of upto 1 time Quantum

Round Robin (RR)

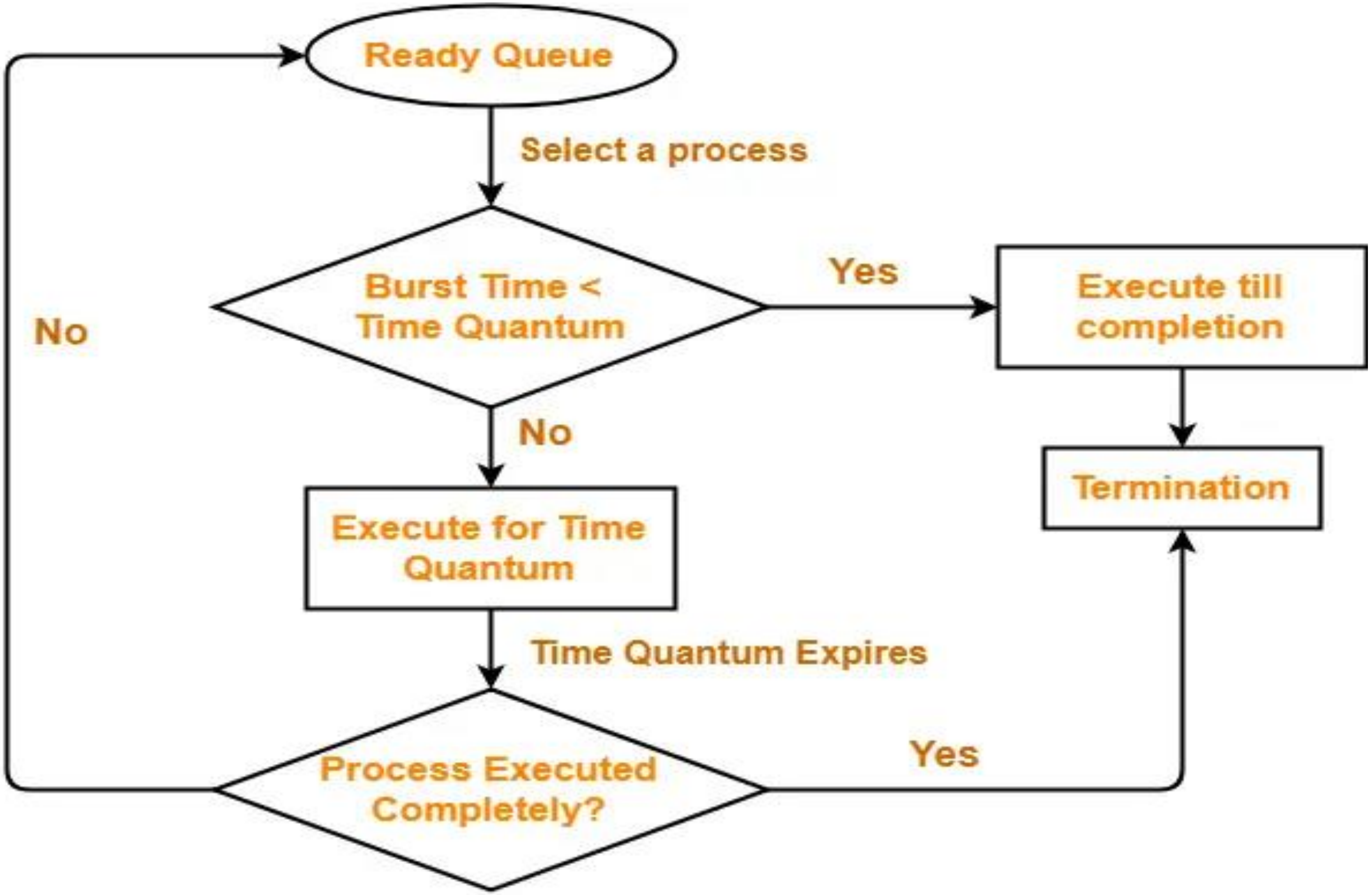
How?

- Ready Queue uses FIFO order
- New processes 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 and dispatches the process.

Round Robin (RR)

How?(contd.)

- Two scenarios:
  - The process may have CPU burst < 1 TQ ,
    - the process will itself release the CPU voluntarily
  - CPU burst>1 TQ,
    - the timer will go off,
    - **causing an interrupt to the OS,**
    - Context switch will be done, process will be put at the tail of the ready queue.



Round Robin Scheduling

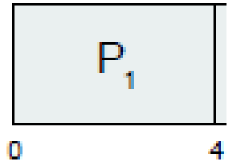
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

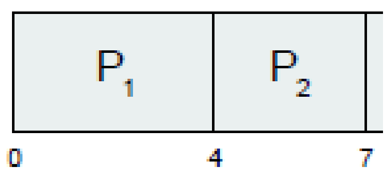
- The Gantt chart is:



Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

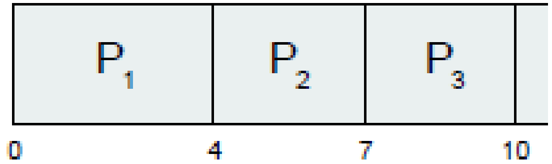
- The Gantt chart is:



Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

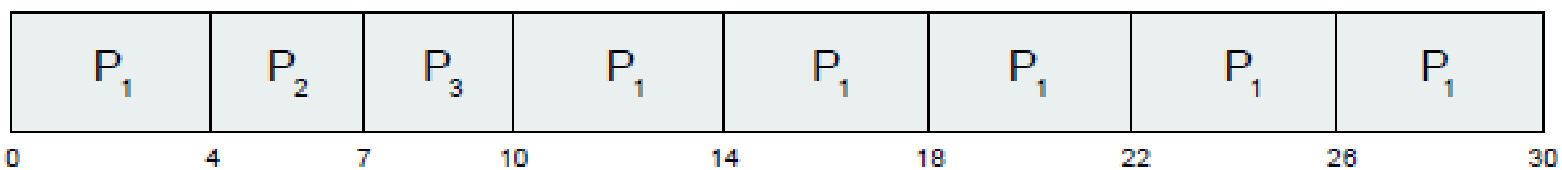
- The Gantt chart is:



Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

Waiting Time –

- P1=(0 + (10-4))=6
- P2=4
- P3=7

AWT=6+4+7/3=17/3

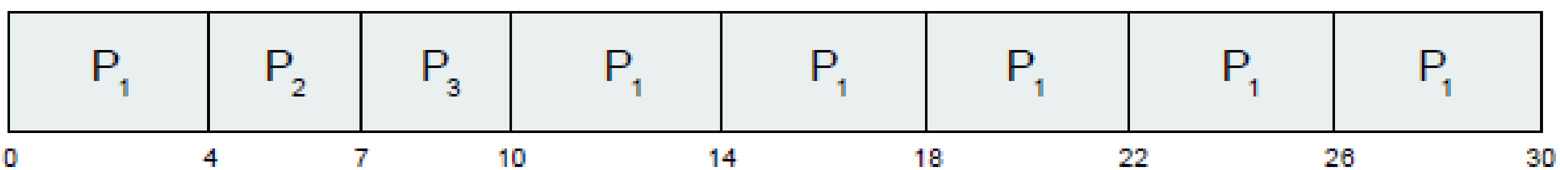
The Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1	
0	4	7	10	14	18	22	26	30

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

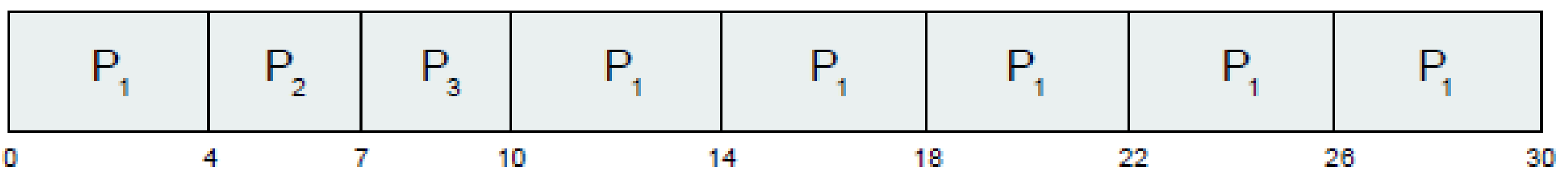


- Typically, higher average turnaround than SJF, but better ***response***

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



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

Response Time, Turnaround Time

- ***P1 =0 ms,30 ms***
- ***P2=4 ms,7ms***
- ***P3=7 ms,10 ms***

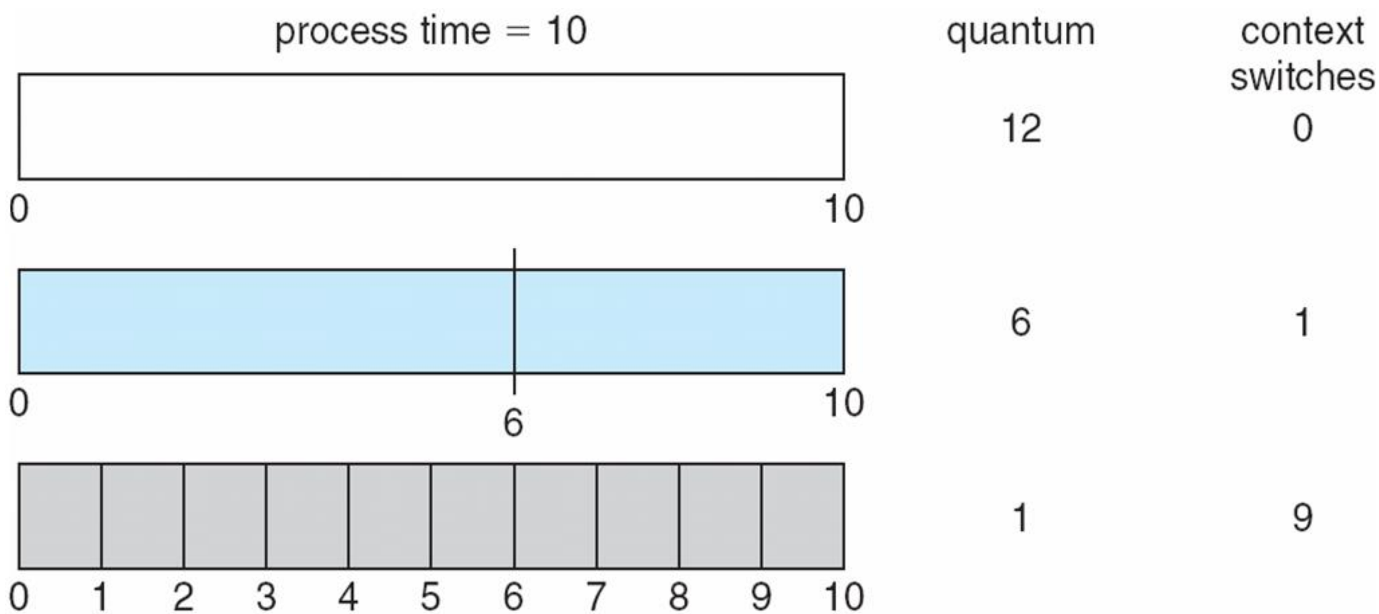
Round Robin (RR)

- RR is preemptive
- 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.**

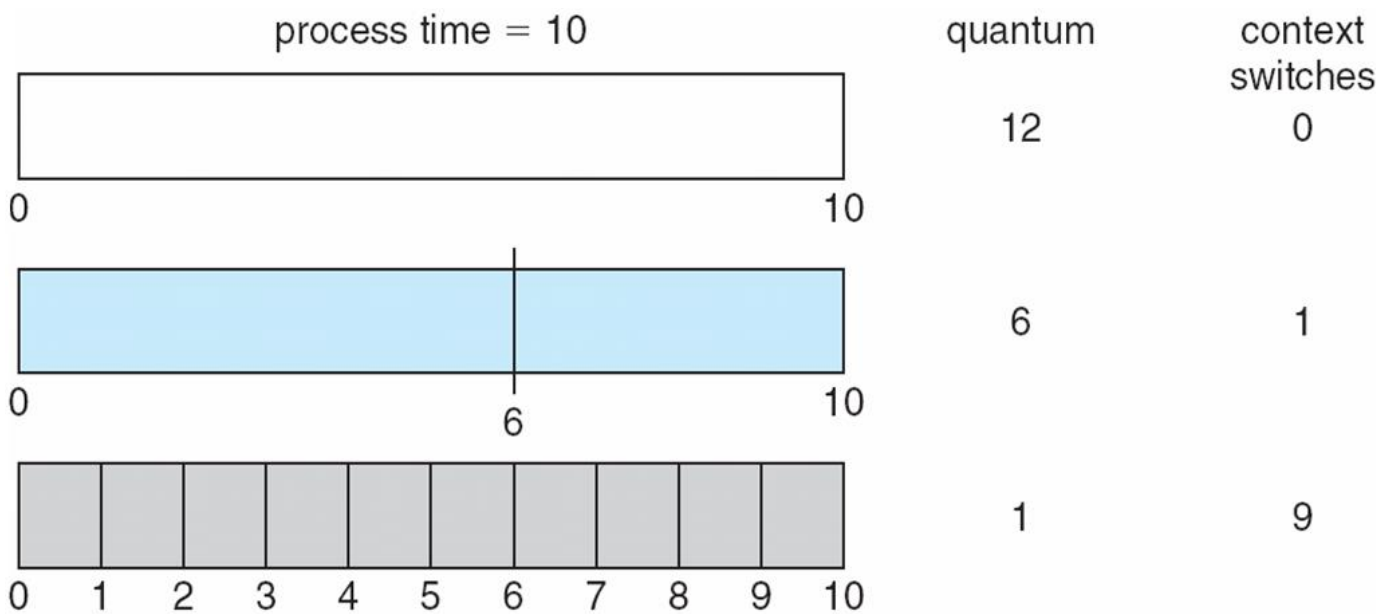
Round Robin (RR)

- Performance
  - *Tq* large $\Rightarrow$ RR is same as FIFO
  - *Tq* small $\Rightarrow$ RR is called processor sharing, appears to user as if each of n processes has its own processor running at 1/n the speed of real processor
  - ***q* must be large with respect to context switch, otherwise overhead is too high**

Time Quantum and Context Switch Time



Time Quantum and Context Switch Time



- The TQ should be large with respect to the context switch time.
  - Tq usually 10ms to 100ms, context switch < 10 usec
- <u>Else much time will be wasted in Context switching.</u>

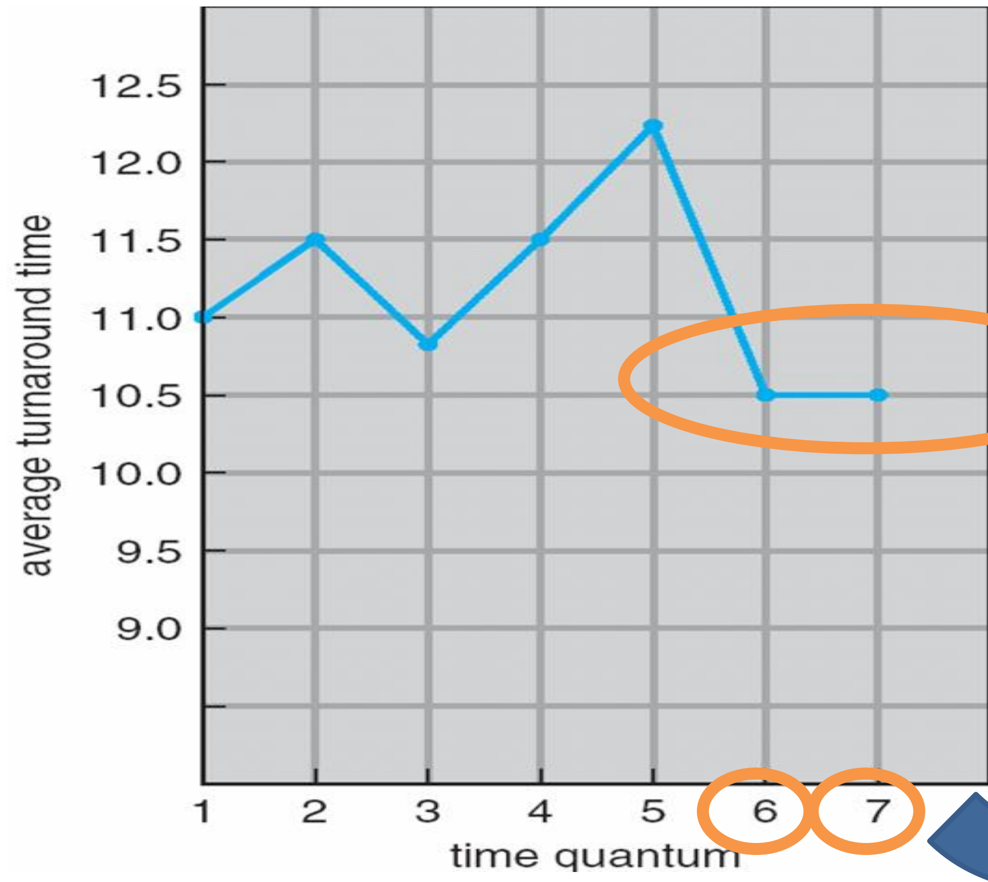
Turnaround Time Varies With The Time Quantum

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

- The Average turnaround time of a set of processes does not necessarily improve as the time quantum increases.
  - The ATT can be improved if most processes finish their next CPU burst in single TQ

- Thumb Rule- "80% of CPU bursts should be shorter than Tq"

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

<u>The ATT can be improved if most processes finish their next CPU burst in single TQ</u>

Thumb Rule- “80% of CPU bursts should be shorter than Tq”

Exercise 2

Consider a set of 6 processes whose arrival time, CPU time needed are given below:

Process	Arrival Time (ms)	Burst Time (ms)
P1	0	8
P2	1	4
P3	2	2
P4	3	1
P5	4	3
P6	5	2

If the CPU scheduling policy is Round Robin. Illustrate the scheduling policy with the help of Gantt chart.

What will be the Average Waiting Time and Average Turnaround time if the scheduling policy is Round Robin. Assume quantum time for Round-Robin as 2 ms.

Soln

Q2 b) RR

Gantt Chart

P1	P2	P3	P1	P4	P5	P2	P6	P1	P5	P1

t=0, 2, 4, 6, 8, 9, 11, 13, 15, 17, 18, 20

tq = 2ms

	AT	BT
P1	0	~~8~~ ~~6~~ ~~4~~ 2
P2	1	~~4~~ ~~2~~ 0
P3	2	~~2~~ 0
P4	3	~~1~~ 0
P5	4	~~3~~ ~~1~~ 0
P6	5	~~2~~ 0

At t = 0, RQ = {(P1, RBT = 8)}
so P1 selected

$\boxed{P1}$
CPU

At t = 1, P2 enters
RQ = {(P2, BT = 4)}

At t = 2, P1 is pre-empted &
P3 also arrives
RQ = { (P2, BT = 4), (P3, BT = 2)}

P2 gets selected

$\boxed{P2}$
CPU

so new ready queue,
RQ = { (P3, BT = 2) (P1, Rem BT = 6) }

At t = 3, P4 arrives
RQ = {(P3, BT = 2) (P1, RBT = 6), (P4, BT = 1)}

Soln

At $t=4$, P_2 is pre-empted
and P_5 also arrives
$RQ = \{(P_3, BT=2), (P_1, RBT=6), (P_4, BT=1), (P_5, BT=3)\}$
P_3 is selected $\boxed{P_3}$ CPU
so new ready queue,
$RQ = \{(P_1, RBT=6)\ (P_4, BT=1)\ (P_5, BT=3)\ (P_2, RBT=2)\}$
At $t=5$, P_6 arrives
$RQ = \{(P_1, RBT=6)\ (P_4, BT=1)\ (P_5, BT=3)\ (P_2, RBT=2)(P_6, BT=2)\}$
At $t=6$, P_3 has finished execution CPU $\boxed{P_1}$
P_1 selected
At $t=8$, P_1 preempted & P_4 selected
$RQ = \{(P_4, BT=1)\ (P_5, BT=3)\ (P_2, RBT=2)\ (P_6, BT=2)(P_1, RBT=4)\}$
CPU $\boxed{P_4}$

KK
At $t=9$, P_4 finishes execution
$RQ = \{(P_5, BT=3), (P_2, RBT=2)\ (P_6, BT=2)\ (P_1, RBT=4)$
P_5 selected CPU $\boxed{P_5}$
At $t=11$, P_5 is pre-empted & P_2 selected CPU $\boxed{P_2}$
$RQ = \{(P_6, BT=2)\ (P_1, RBT=4)\ (P_5, RBT=1)$
At $t=13$, P_2 finishes, P_6 selected CPU $\boxed{P_6}$
$RQ = \{(P_1, RBT=4)\ (P_5, RBT=1)\}$

Soln

At $t=15$, P_6 completes

P_1 is selected $\boxed{P_1}$ CPU

At $t=17$ P_1 is pre-empted, P_5 is selected $\boxed{P_5}$ CPU

$RQ = \{(P_5, RBT=1)\}$

So new Ready Queue $RQ = \{(P_1, RBT=2)\}$

At $t=18$, P_5 completes

P_1 selected

At $t=20$ P_1 completes

Summary

	AT	BT	WT	WT	TAT (complet^n − AT)	TAT
P_1	0	8	$0+(6-2)+(15-8)+(18-17)$	12	$(20-0)$	20
P_2	1	4	$(2-1)+(11-4)$	8	$(13-1)$	12
P_3	2	2	$(4-2)$	2	$(6-2)$	4
P_4	3	1	$(8-3)$	5	$(9-3)$	6
P_5	4	3	$(9-4)+(17-11)$	11	$(18-4)$	14
P_6	5	2	$(13-5)$	8	$(15-5)$	10

$AWT = \frac{46}{6} = 7.66\ ms$

$ATAT = \frac{66}{6} = 11\ ms$

Breakup

- **Gantt Chart=1 M**
- **Average Waiting Time=2 M**
- **Average Turnaround time=2 M**

Pre-emptive Algorithms

Shortest Remaining Time First (SRTF)

- In the Shortest Remaining Time First (SRTF) scheduling algorithm,
  - The process with the smallest amount of time remaining until completion is selected to execute.

Shortest Remaining Time First (Preemptive SJF): Example

PROCESS	DURATION	ORDER	ARRIVAL TIME
P1	9	1	0
P2	2	2	2

Shortest Remaining Time First (Preemptive SJF): Example

PROCESS	DURATION	ORDER	ARRIVAL TIME
P1	9	1	0
P2	2	2	2

P1 waiting time: 0 + (4-2-0) = 2
P2 waiting time: (2-2)=0
The average waiting time(AWT): (0 + 2) / 2 = 1

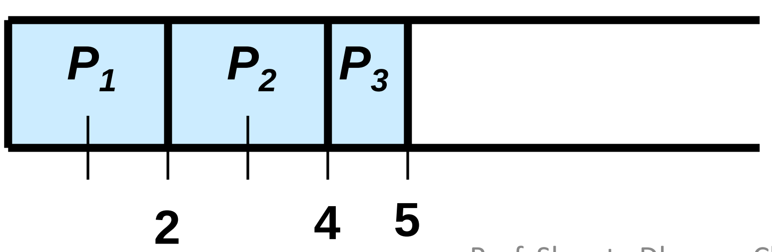
Example for Preemptive SJF (SRTF)

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

Example for Preemptive SJF (SRTF)

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

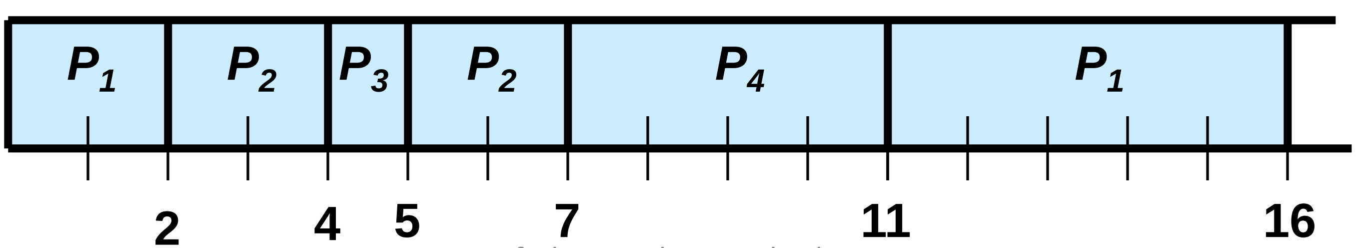
- Time 0 – P_1 gets the CPU Ready = [(P_1,7)}
- Time 2 – P_2 arrives – CPU has **P_1 with rem. time=5**,
- Ready = [(P_2,4)] -P_2 gets the CPU, P1 is preempted
- Ready = [(P_1,5)]
- Time 4 – P_3 arrives – CPU has **P_2 with rem. time = 2**,
- Ready = [(P_1,5),(P_3,1)] – P_3 gets the CPU, P2 is preempted
- Ready = [(P_1,5),(P2,2)}



Example for Preemptive SJF (SRTF)

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

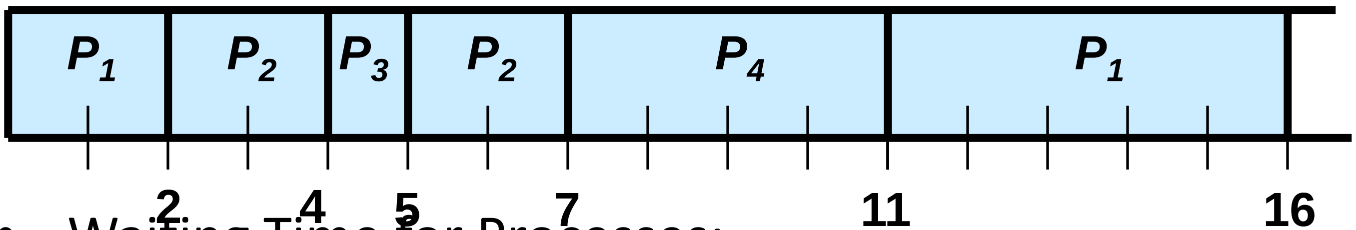
- Ready = [(P_1,5),(P2,2)]
- Time 5 – P_3 completes and P_4 arrives –
- Ready = [(P_1,5),(P_2,2),(P_4,4)] – P_2 gets the CPU
- Time 7 – P_2 completes –
- Ready = [(P_1,5),(P_4,4)] – P_4 gets the CPU
- Time 11 – P_4 completes, P_1 gets the CPU



Example for Preemptive SJF (SRTF)

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)



- Waiting Time for Processes:
- P1=(0+(11-2))=9,
- P2=((2-2)+(5-4))=1,
- P3=(4-4)=0,
- P4=(7-5)=2
- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Example of Shortest-remaining-time-first

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

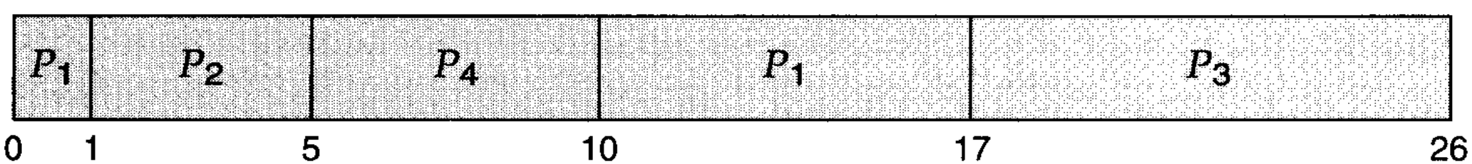
????

Try solving this………..

Example of Shortest-remaining-time-first

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

- Time 0 – P_1 gets the CPU   Ready = [(P_1,8)]
- Time 1 – P_2 arrives – CPU has P_1 with rem. time=7, Ready = [(P_2,4)] – P_2 gets the CPU
- Time 2 – P_3 arrives – CPU has P_2 with rem. time = 3, Ready = [(P_1,7)(P_3,9)] – P2 continues with the CPU
- Time 3- P4 arrives- CPU has P2 with rem. time=2, Ready =[(P_1,7)(P_3,9)(P4,5)] – P2 continues with the CPU
- After P2 finishes, then P4 executes , then P1 , finally P3
- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 msec

Estimating the Length of Next CPU Burst

- Problem with SJF: It is very difficult to know exactly the length of the next CPU burst.
- Idea: Based on the observations in the recent past, we can try to *predict*.

Estimating the Length of Next CPU Burst

Exponential averaging:

- nth CPU burst = t_n
- the average of all past bursts τ_n,
- using a weighting factor $0 <= \alpha <= 1$,
- the next CPU burst is: $\tau_{n+1} = \alpha\ t_n + (1 - \alpha)\ \tau_n$.

Estimating the Length of Next CPU Burst

- *This formula defines an Exponential average of the measured lengths of previous CPU bursts.*
- t_n= nth CPU burst , contains most recent information,
- τ_n =the average of all past bursts, contains past history,
- α controls the relative weight of recent and past history in our prediction.
- We expect that the next CPU burst will be similar in length to the previous ones.
- $\tau_{n+1} = \alpha\ t_n + (1- \alpha)\ \tau_n.$

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not matter
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the most recent CPU burst matters
  - history is assumed to be old and irrevelant.

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

Estimating the Length of Next CPU Burst

- More commonly $\alpha=1/2$, so recent history and past history is equally weighted
- The initial τ_0 can be defined as a constant or as an overall system average

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

Examples of Exponential Averaging

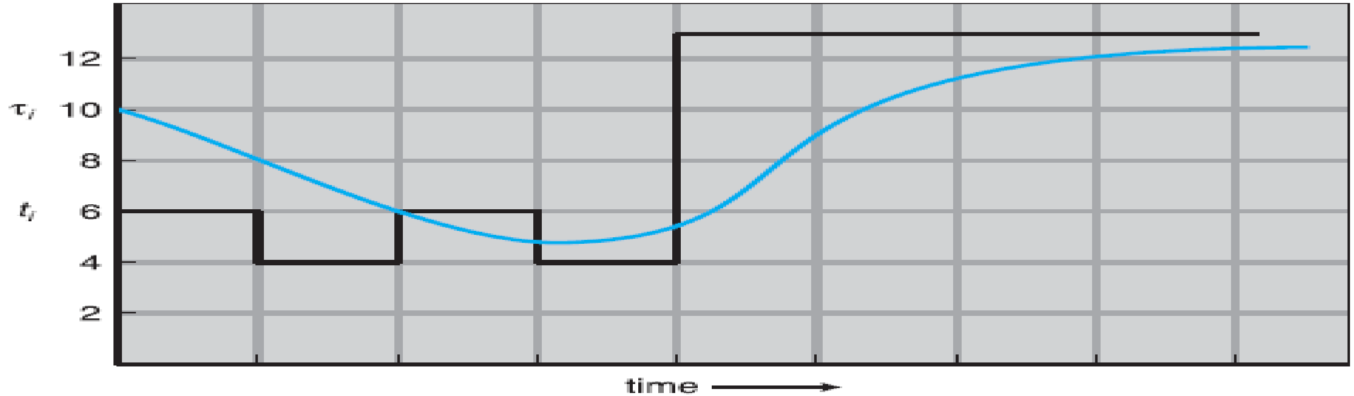
- To Understand the behavior of the exponential average,
- We can expand the formula for τ_{n+1} by substituting for τ_n ,If we expand the formula, we get:

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

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

Estimating the Length of Next CPU Burst

- *Figure shows exponential average with* $\alpha=1/2$ and $\tau_0=10$



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

To understand the behavior of the exponential average, we can expand the formula for τ_{n+1} by substituting for τ_n, to find

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

Priority Scheduling

- A priority number (integer) is associated with each process

- **The CPU is allocated to the process with the highest priority** (smallest integer $\equiv$ highest priority)

Priority Scheduling

- SJF is a special case of priority scheduling:
  - process priority = the *inverse of remaining CPU time*
  - **The larger the CPU burst, the lower the priority and vice versa**

- **Equal priority processes are scheduled in FCFS order**

  - FCFS can be used to break ties.

Priority Scheduling

- We discuss in terms of High, low priority
- Priority are fixed range of numbers such as 0 to 7 or 0 to 4,095.
  - No agreement on whether 0 is highest /lowest.
  - Some systems use lower numbers to represent low priority other use low numbers for high priority.
  - Here , we use low numbers for high priority

Priority Scheduling

- Priorities can be defined:
  – Internally
  – externally

Priority Scheduling

- Internally defined-
  - Use Some measurable quantities
  - Eg- **time limits, memory requirements, the number of open files, ratio of average I/O burst to average CPU burst have been used**

- Externally defined-
  - Set by criteria that are external to the OS
  - **Importance of the process**
  - **Type and amount of fund being paid for computer use**
  - **The Department sponsoring the work**
  - **Often political factors**

Priority Scheduling

- Priority can be :
  - **pre-emptive**
  - **non pre-emptive**

- When a process arrives at the ready queue,
  - **the priority is compared with priority of the current running process.**

Priority Scheduling

Scenario:

- If the priority of the newly arrived process is higher than the priority of the currently running process.

Characteristics:

- A preemptive priority scheduling algorithm will
  - preempt the CPU
- **A non-preemptive priority scheduling algorithm will**
  - **simply put the new process at the head of the ready queue ,**

Priority Scheduling

- Problem ≡ **Starvation/Indefinite blocking**

- Solution ≡ **Aging**

Priority Scheduling

- Problem ≡ **Starvation/Indefinite blocking**

- Low priority processes may never execute
  - Leave some low priority processes waiting indefinitely for the CPU

Priority Scheduling

Solution ≡ **Aging** – as time progresses increase the priority of the process that wait in the system for a long time.

- For eg- If priorities range from 127 (low) to 0 (high), decrement the priority of a waiting process by 1 every 15 minutes.

Example of Non Pre-emptive Priority Scheduling

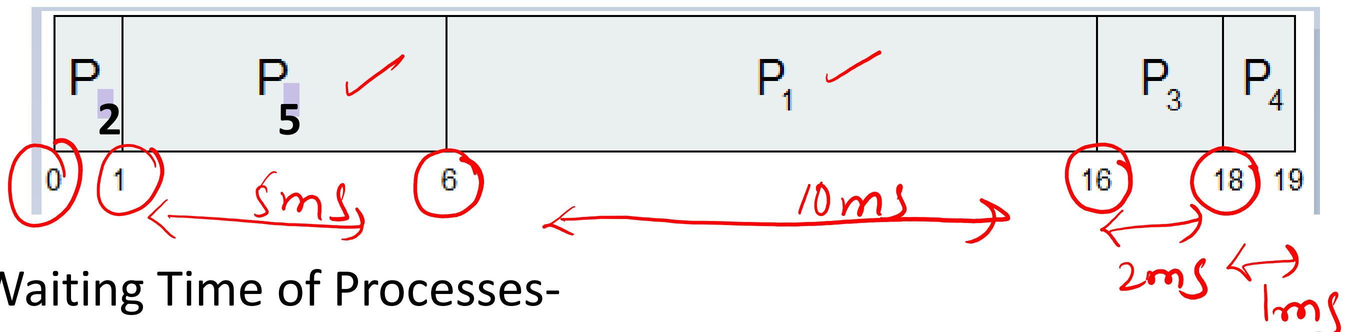
Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Example of Non Pre-emptive 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

Gantt

- Priority scheduling Gantt Chart



Waiting Time of Processes-

P1=6

P2=0

P3=16

P4=18

P5=1

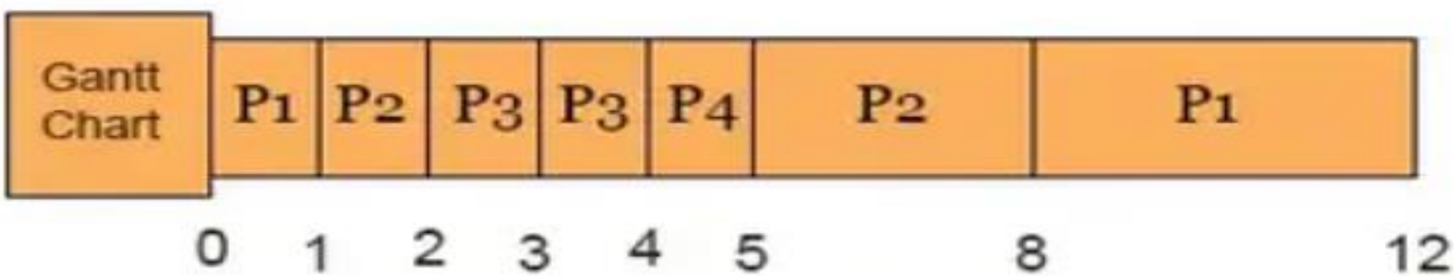
- Average waiting time = =(6+0+16+18+1)/5=41/5=8.2 msec

Pre-emptive Priority Algorithm

Process	Arrival Time	Burst Time	Priority
P1	0	5	10
P2	1	4	20
P3	2	2	30
P4	4	1	40

Pre-emptive Priority Algorithm

Process	Arrival Time	Burst Time	Priority
P1	0	5	10
P2	1	4	20
P3	2	2	30
P4	4	1	40



Pre-emptive Priority Algorithm

Process	Arrival Time	Burst Time	Priority	Completion Time	Turnaround Time	Waiting Time	Response Time	Avg. Time Calculations
P1	0	5	10	12	12	7	0	Average Waiting Time = (7+3+0+0)/4 = 2.5
P2	1	4	20	8	7	3	0	Average Turnaround Time = (12+7+2+1)/4 =5.5
P3	2	2	30	4	2	0	0	
P4	4	1	40	5	1	0	0	

Gantt Chart	P1	P2	P3	P3	P4	P2	P1

0 1 2 3 4 5 8 12

Preemptive Priority Scheduling

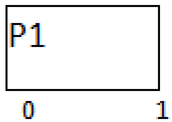
Process Id	Priority	Arrival Time	Burst Time
1	2	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10	5	15
7	9	15	8

Preemptive Priority Scheduling

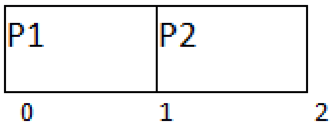
Process Id	Priority	Arrival Time	Burst Time
1	2	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10	5	15
7	9	15	8

At time=0, P1 arrives with the burst time of 1 units and priority 2.
Since no other process is available hence this will be scheduled till next job arrives or its completion (whichever is lesser).

Ready Queue={(P1,BT=1,PR=2)}



At time=1, P2 arrives.
P1 terminates
No other process is available at this time hence the Operating system has to schedule P2 regardless of the priority assigned to it.
Ready Queue={(P2,BT=7,PR=6)}



Preemptive Priority Scheduling

Process Id	Priority	Arrival Time	Burst Time
1	2	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10	5	15
7	9	15	8

At Time=2
P3 arrives , the priority of P3 is higher to P2.
Ready Queue={(P3,BT=3,PR=3)}

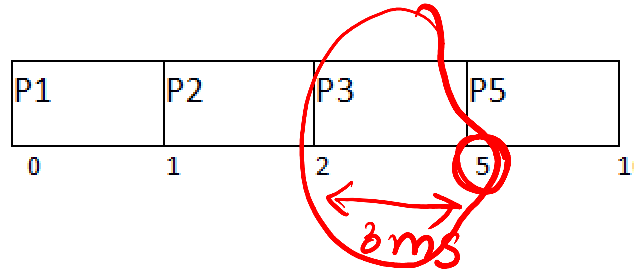
P2 with Remaining Time=6,PR=6
P2 is Pre-empted
P3 is selected
Ready Queue={(P2,Rem BT=6,PR=6)}

P1	P2	P3

0 1 2 5

Preemptive Priority Scheduling

Process Id	Priority	Arrival Time	Burst Time
1	2	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10	5	15
7	9	15	8



During the execution of P3, three more processes P4, P5 and P6 becomes available.

earlier preempted process

Ready Queue={(P2,Rem BT=6,PR=6), (P4, BT=6,PR=5), (P5,BT=5,PR=4), (P6,BT=15,PR=10)}

Since, all these three have the priority lower to P3,

No pre-emption, P3 continues

At t=5, P3 terminates

Preemptive Priority Scheduling

Process Id	Priority	Arrival Time	Burst Time
1	2	0	1 √
2	6	1	7
3	3	2	3 √
4	5	3	6 √
5	4	4	5 √
6	10	5	15
7	9	15	8

last
late

P1	P2	P3	P5	P4

0 1 2 5 10 16

Ready Queue={(P2,Rem BT=6,PR=6), (P4, BT=6,PR=5), (P5,BT=5,PR=4), (P6,BT=15,PR=10)}

P5 will be scheduled with the priority highest among the available processes.

During the execution of P5, all the processes got available in the ready queue. At this point, the algorithm will start behaving as Non Preemptive Priority Scheduling.

Ready Queue={(P2,Rem BT=6,PR=6), (P4, BT=6,PR=5), (P6,BT=15,PR=10)}

At t=10, P5 terminates

P4 selected with highest priority in ready queue and will be executed till the completion.

Ready Queue={(P2,Rem BT=6,PR=6), (P6,BT=15,PR=10)}

At t=15, P7 arrives, with lower priority, P4 continues

At t=16, P4 terminates

Preemptive Priority Scheduling

Process Id	Priority	Arrival Time	Burst Time
1	2	0	1 √
2	6	1	7 √
3	3	2	3 √
4	5	3	6 √
5	4	4	5 √
6	10	5	15
7	9	15	8

At t=16

Ready Queue={(P2,Rem BT=6,PR=6), (P6,BT=15,PR=10) (P7,BT=8,PR=9)}

P2 is selected, **as it has highest priority**

P2 terminates

Ready Queue={(P6,BT=15,PR=10),(P7,BT=8,PR=9)}

P1	P2	P3	P5	P4	P2

0 1 2 5 10 16 22

Preemptive Priority Scheduling

Process Id	Priority	Arrival Time	Burst Time
1	2	0	1 √
2	6	1	7 √
3	3	2	3 √
4	5	3	6 √
5	4	4	5 √
6	10	5	15 √
7	9	15	8 √

Ready Queue={(P6,BT=15,PR=10),(P7,BT=8,PR=9)}

P7 will be selected

P1	P2	P3	P5	P4	P2	P7	
0	1	2	5	10	16	22	30

The only remaining process is P6 with the least priority, will be executed at the last.

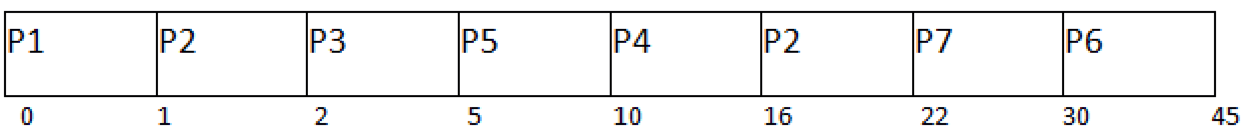
P1	P2	P3	P5	P4	P2	P7	P6	
0	1	2	5	10	16	22	30	45

Preemptive Priority Scheduling

Process Id	Priority	Arrival Time	Burst Time	Completion Time	Turn around Time	Waiting Time
1	2	0	1	1	1	0
2	6	1	7	22	21	14
3	3	2	3	5	3	0
4	5	3	6	16	13	7
5	4	4	5	10	6	1
6	10	5	15	45	40	25
7	9	6	8	30	24	16

Turnaround Time = Completion Time - Arrival Time

Waiting Time = Turn Around Time - Burst Time or Sum of times spent in Ready queue



Avg Waiting Time = (0+14+0+7+1+25+16)/7 = 63/7 = 9 units

https://www.javatpoint.com/os-preemptive-priority-scheduling

Multilevel Queue

What?

- Processes are classified into different groups.
- Common division is
  - **foreground** (interactive)
  - **background** (batch)

Multilevel Queue

Why?

- These Processes
  - have different response time requirements ,
  - so have different scheduling needs.
  - Foreground process have higher priority over background processes

Multilevel Queue

How?

- Multi Level Queue Scheduling Algorithm(MQSA) partitions **<u>Ready queue into separate queues</u>**, eg:

- **Process permanently assigned to a given queue based on some property as:**
  - Process priority
  - Process type
  - Memory size

Multilevel Queue

- Each queue has its own scheduling algorithm:
  - **foreground – RR**
  - **background – FCFS**

- Scheduling must be done between the queues:
  - **Fixed priority scheduling;**
  - **Time Slicing**

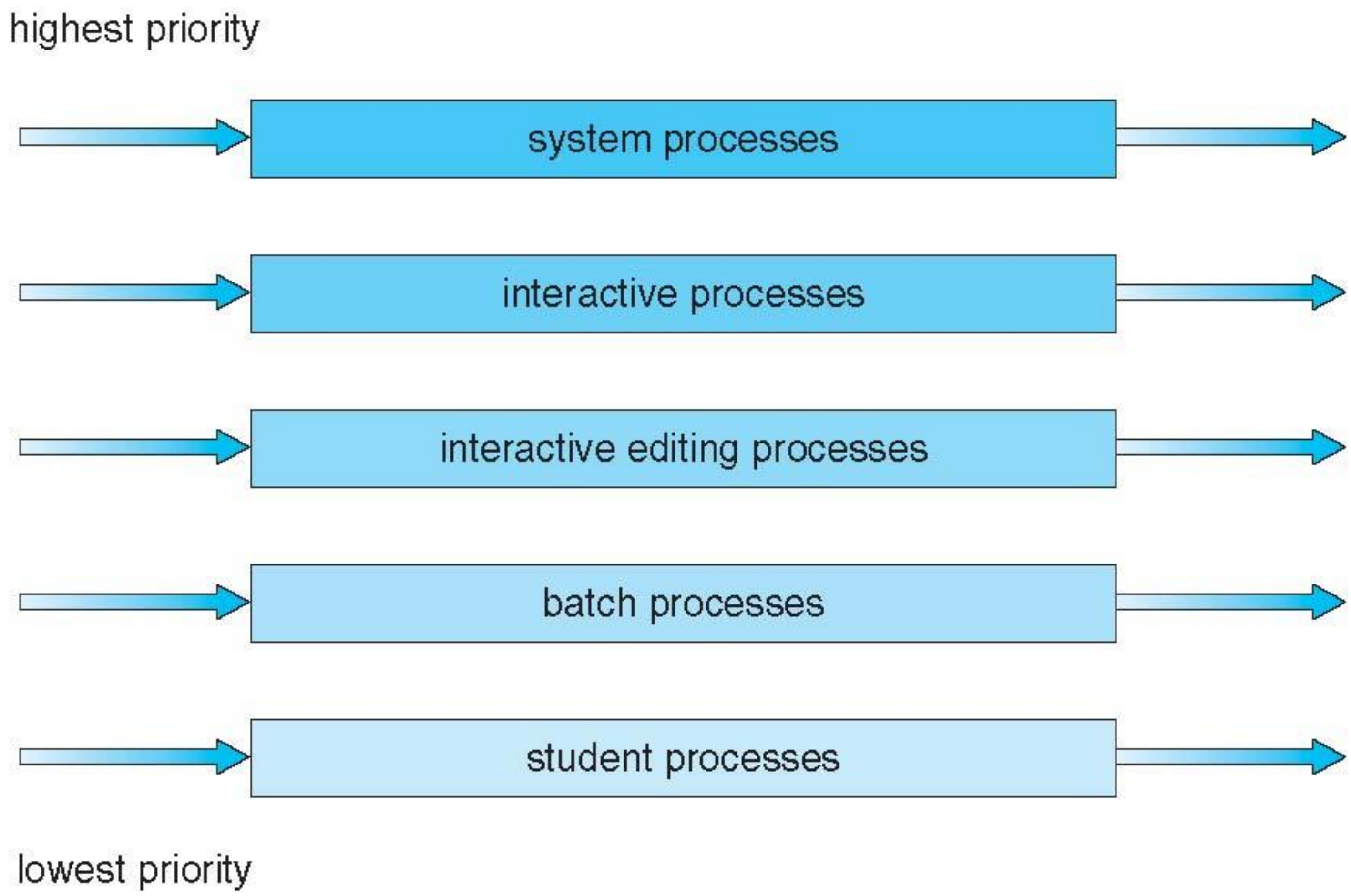
Multilevel Queue

- Fixed priority scheduling;
  - Foreground Queue has absolute priority over Background queue
  - Serve all from foreground then from background
  - Possibility of starvation.

Multilevel Queue

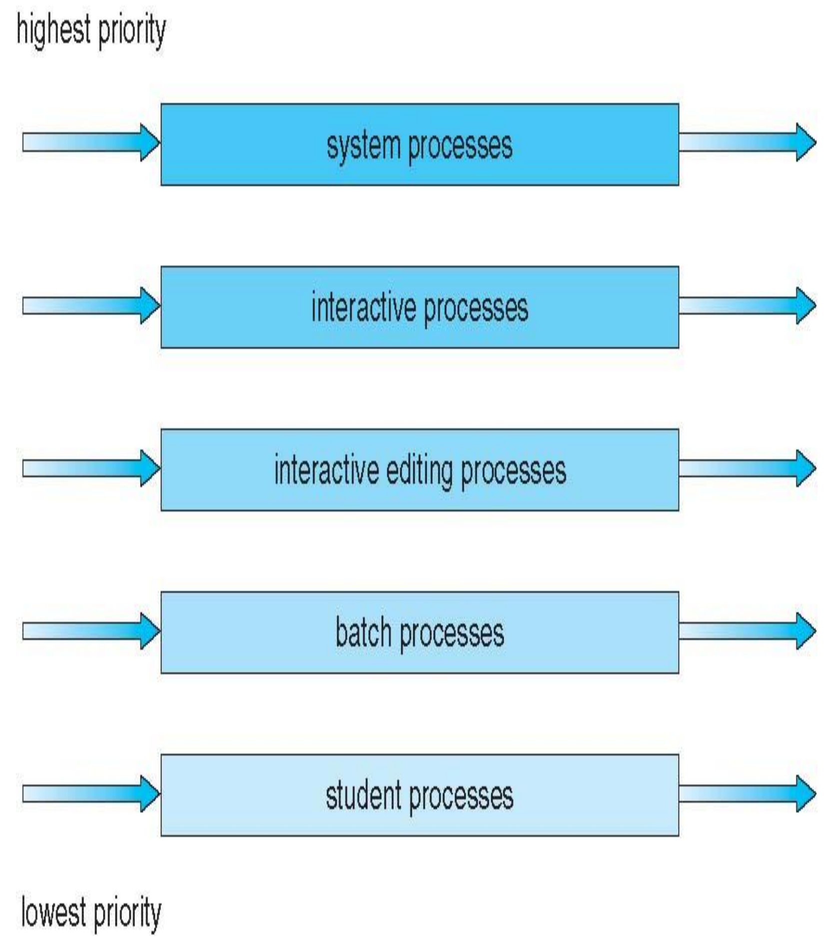
- Time slice between the queue
  - each queue gets a certain amount of CPU time which it can schedule amongst its processes;
  - For eg, 80% of CPU time to foreground in RR
  - 20% to background in FCFS

Multilevel Queue Scheduling



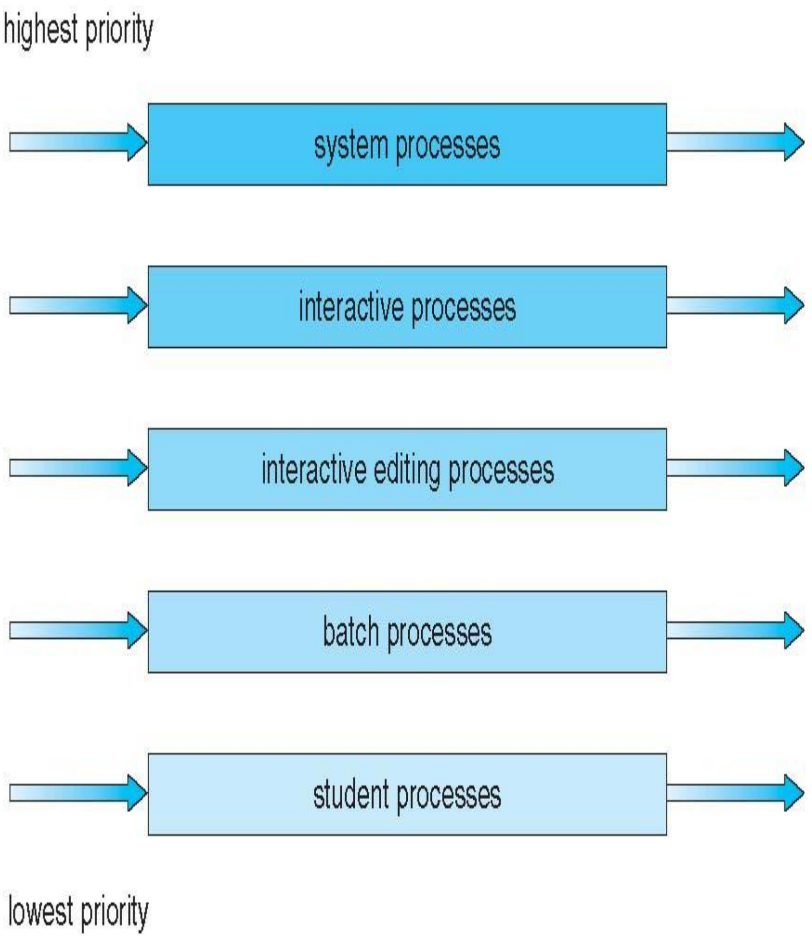
Multilevel Queue- Possibility of starvation

- MQSA with 5 queues
- **Each queue has absolute priority over lower priority queues.**
- No process in batch queue could run
  - unless the queues from system, interactive, interactive editing queue were all empty.



Multilevel Queue- Possibility of starvation

- If an interacting editing process entered the ready queue while a batch process was running,
  - the batch process would be preempted.



Multilevel Queue

- Process do not move between queues.
  - Since processes do not change their foreground or background nature.
- Advantage-
  - Low scheduling overhead
- Disadvantage –
  - Being Inflexible

Multilevel Feedback Queue

- A process can move between the various queues

- How?

Multilevel Feedback Queue

How?

- The idea is to Separate Processes with different **CPU burst characteristic.**
  - **Process uses too much CPU time,**
    - **will be moved to a lower priority queue.**
  - **This scheme leaves I/O bound and interactive processes**
    - **in the higher priority queues.**

Multilevel Feedback Queue

Starvation=

- A process that waits too long in a lower priority queue

Solution=

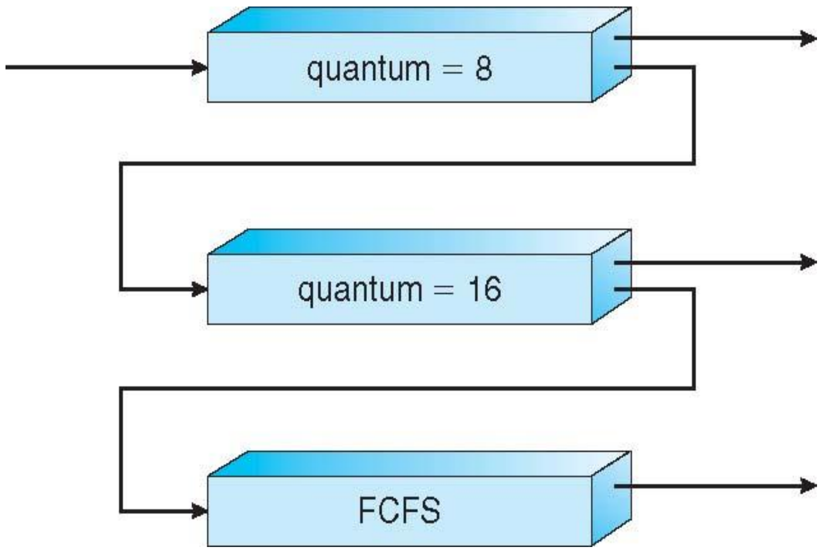
- Aging can be implemented this way to prevent starvation
  - may be moved to a higher priority queue.

Multilevel Feedback Queue

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - <u>method used to determine when to upgrade a process</u>
  - <u>method used to determine when to demote a process</u>
  - <u>method used to determine which queue a process will enter when that process needs service</u>
- Most general scheme but most complex

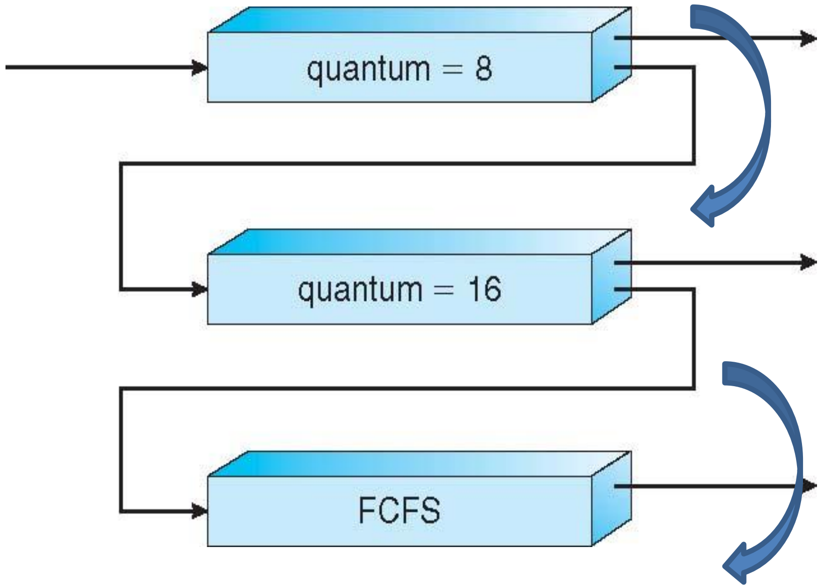
Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS



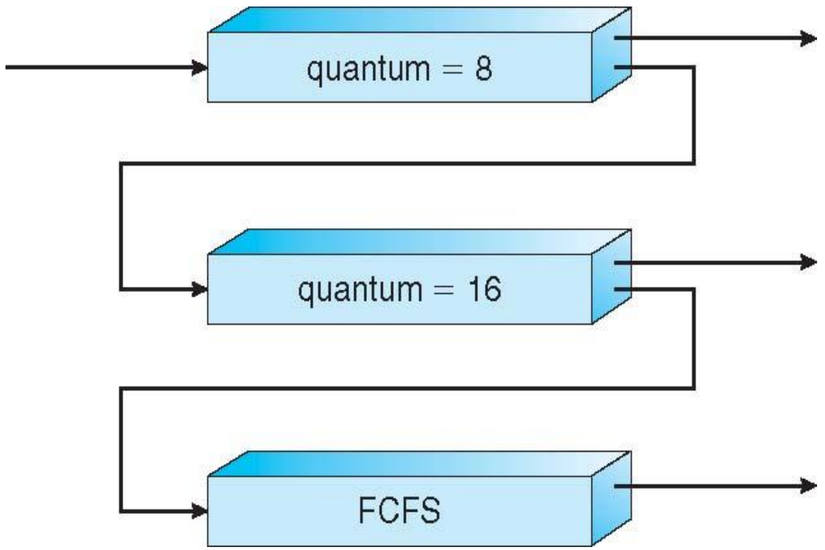
Example of Multilevel Feedback Queue

- Scheduler first executes all processes in Queue 0,
  - Only when Queue0 is empty,
    - will it execute processes in Queue1.
  - Queue2 will be executed
    - only when Queue 0 and Queue 1 is empty



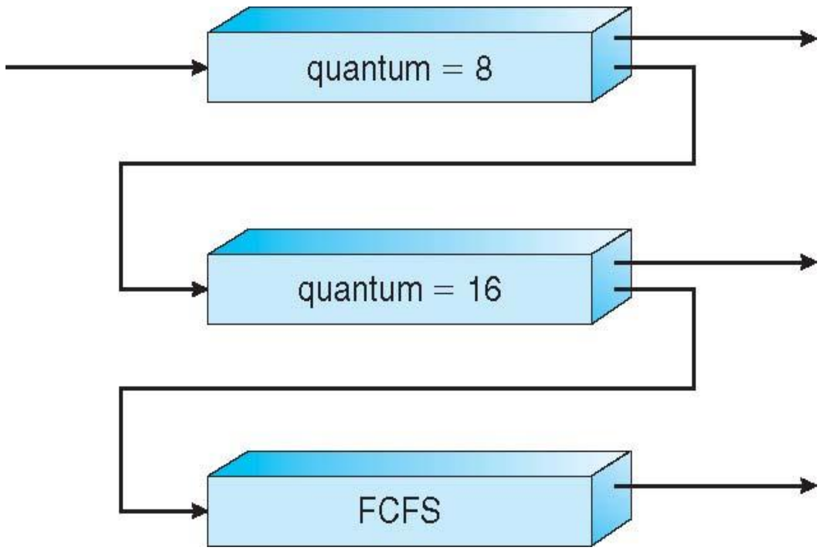
Example of Multilevel Feedback Queue

- A process that arrives for Q1
  - will preempt a process in Q2
- A process that arrives for Q0
  - will preempt a process in Q1



Example of Multilevel Feedback Queue

- Scheduling
  - A new job enters queue Q_0 which is served RR
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to tail of queue Q_1

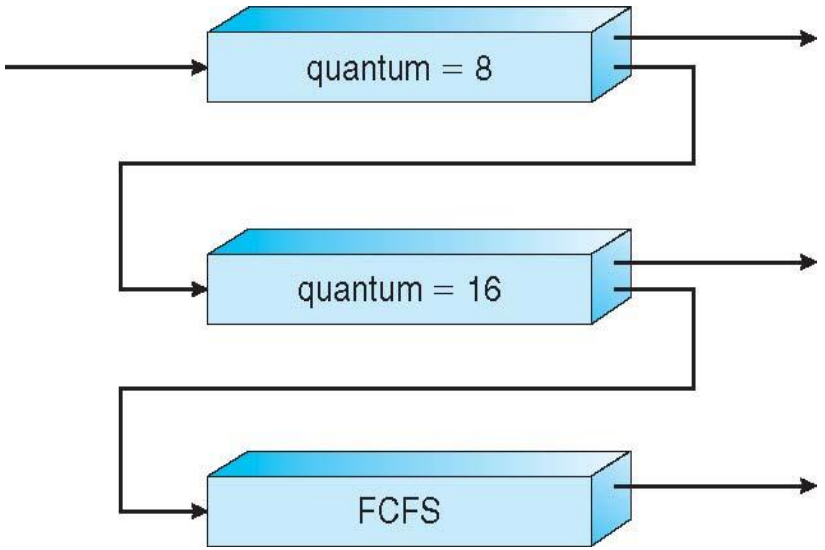


Example of Multilevel Feedback Queue

- Scheduling

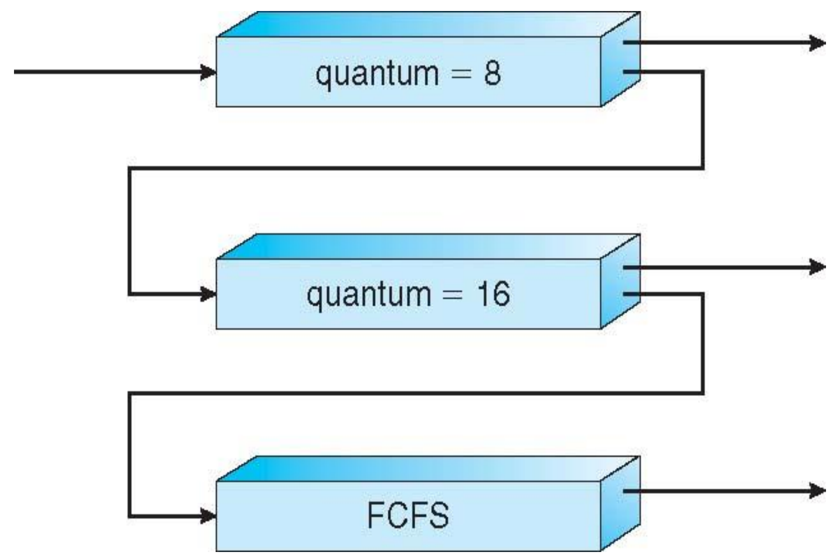
  – If Q0 is empty, the process at the head of Q1 is given time quantum of 16 milliseconds

    - At Q_1 job again receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Example of Multilevel Feedback Queue

- Scheduling
  - Processes with CPU burst of 8 ms or less are given high priority
  - Processes with CPU burst >8 but <24 ms are also served quickly with less priority
  - Long processes automatically sink to Queue 2



Advantages of Multilevel Scheduling:

1) You can use multilevel queue scheduling to apply different scheduling methods to distinct processes.
2) It will have low overhead in terms of scheduling.

Disadvantages of Multilevel Scheduling:

1) There is a risk of starvation for lower priority processes.
2) It is rigid in nature.

Advantages of Multilevel Feedback Queue Scheduling:

1) It is more flexible.
2) It allows different processes to move between different queues.
3) **It prevents starvation by moving a process that waits too long for the lower priority queue to the higher priority queue.**

Disadvantages of Multilevel Feedback Queue Scheduling:

1) It produces more CPU overheads.
2) It is the most complex algorithm.

Difference b/w MLQ and MLFQ

Multilevel queue scheduling (MLQ)	Multilevel feedback queue scheduling (MLFQ)
It is queue scheduling algorithm in which ready queue is partitioned into several smaller queues and processes are assigned permanently into these queues. The processes are divided on basis of their intrinsic characteristics such as memory size, priority etc.	In this algorithm, ready queue is partitioned into smaller queues on basis of CPU burst characteristics. The processes are not permanently allocated to one queue and are allowed to move between queues.
In this algorithm queue are classified into two groups, first containing background processes and second containing foreground processes. 80% CPU time is given to foreground queue using Round Robin Algorithm and 20% time is given to background processes using First Come First Serve Algorithm.	Here, queues are classified as higher priority queue and lower priority queues. If process takes longer time in execution it is moved to lower priority queue. Thus, this algorithm leaves I/O bound and interactive processes in higher priority queue.
The priority is fixed in this algorithm. When all processes in one queue get executed completely then only processes in other queue are executed. Thus, starvation can occur.	The priority for process is dynamic as process is allowed to move between queue. A process taking longer time in lower priority queue can be shifted to higher priority queue and vice versa. Thus, it prevents starvation.
Since, processes do not move between queues, it has low scheduling overhead and is inflexible.	Since, processes are allowed to move between queues, it has high scheduling overhead and is flexible.

Linux Scheduling

Linux Scheduling Prior to kernel Version 2.5

- Prior to kernel version 2.5, ran variation of traditional UNIX scheduling algorithm

- Two problems with the traditional UNIX scheduler are
  - **it does not provide adequate support for SMP systems**
  - **it does not scale well as the number of tasks on the system grows.**

Linux Scheduling Through Version 2.5

- Version 2.5 moved to constant order $O(1)$ scheduling time
- Regardless of the number of tasks on the system.
- The new scheduler also provides
  - **increased support for SMP,**
  - **including processor affinity and load balancing,**
  - **as well as providing fairness and support for interactive tasks.**

Linux Scheduling Through Version 2.5

- The Linux scheduler is
  - a preemptive priority-based algorithm

Linux Scheduling Through Version 2.5

- The Linux scheduler is
  - a preemptive priority-based algorithm
  - with two separate priority ranges:
  - a real-time range from 0 to 99 and
  - a nice value ranging from 100 to 140.

Figure 5.15 The relationship between priorities and time-slice length.

Linux Scheduling Through Version 2.5

- These two ranges map into a global priority scheme wherein
  - numerically lower values indicate higher priorities.

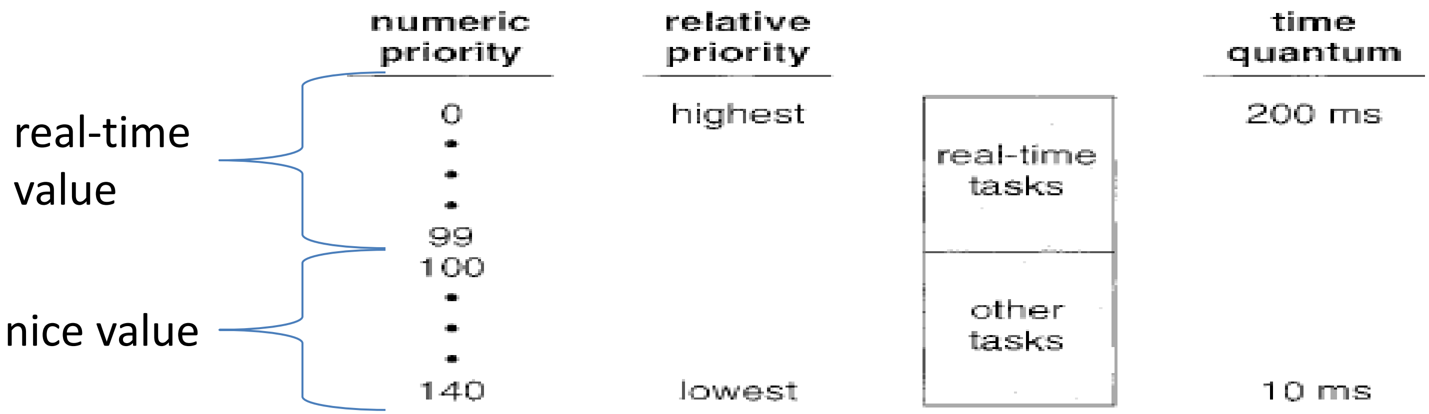


Figure 5.15 The relationship between priorities and time-slice length.

Linux Scheduling Through Version 2.5

- Unlike schedulers for many other systems, including
  - Solaris
  - Windows XP
- **<u>Linux assigns</u>**
  - **<u>higher-priority tasks longer time quanta and</u>**
  - **<u>lower-priority tasks shorter time quanta.</u>**

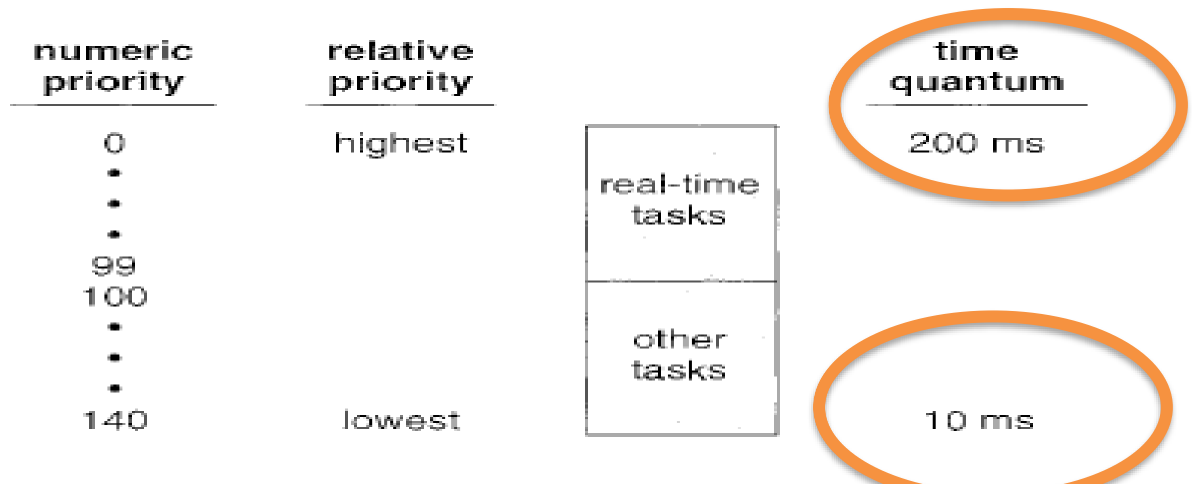


Figure 5.15 The relationship between priorities and time-slice length.

Linux Scheduling Through Version 2.5

- A runnable task is considered eligible for execution on the CPU
  - as long as it has time remaining in its time slice.

- When a task has exhausted its time slice,
  - it is considered expired and
  - is not eligible for execution again
  - until all other tasks have also exhausted their time quanta.

Linux Scheduling Through Version 2.5

- The kernel maintains a list of all runnable tasks
  - in a runqueue data structure.

Linux Scheduling Through Version 2.5

- Because of its support for SMP,
  - Each processor maintains
    - its own runqueue and
    - schedules itself independently.

Linux Scheduling Through Version 2.5

- Each runqueue contains **two priority arrays**:
  - **Active**
  - **Expired**

Linux Scheduling Through Version 2.5

- The active array
  - contains all tasks with **time remaining in their time slices,**
- The expired array
  - contains all expired tasks.

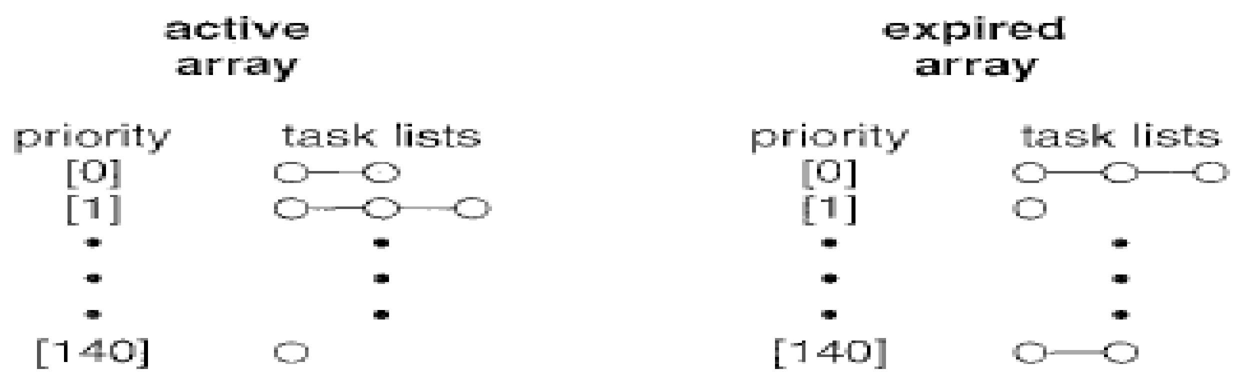


Figure 5.16 List of tasks indexed according to priority.

Linux Scheduling Through Version 2.5

- Each of these priority arrays contains
  - a list of tasks indexed according to priority

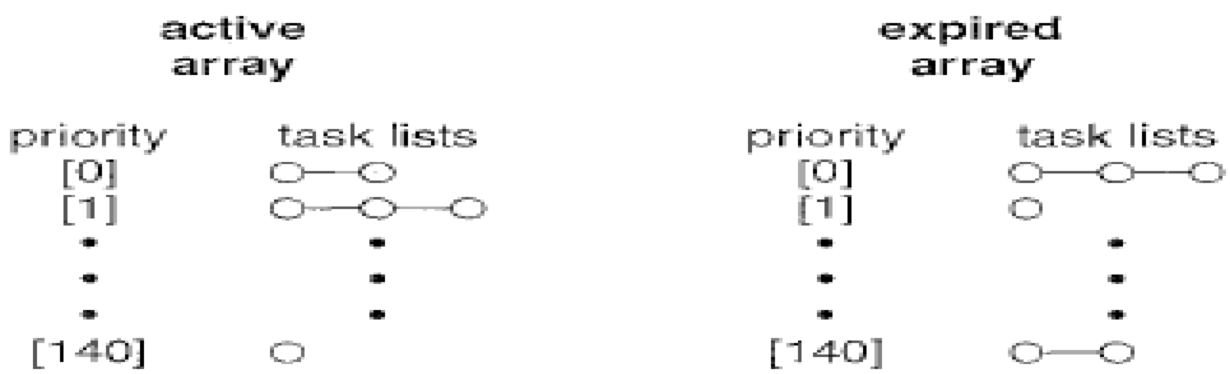


Figure 5.16 List of tasks indexed according to priority.

Linux Scheduling Through Version 2.5

- The scheduler chooses the task
  - **with the highest priority from the active array**
  - **for execution on the CPU.**

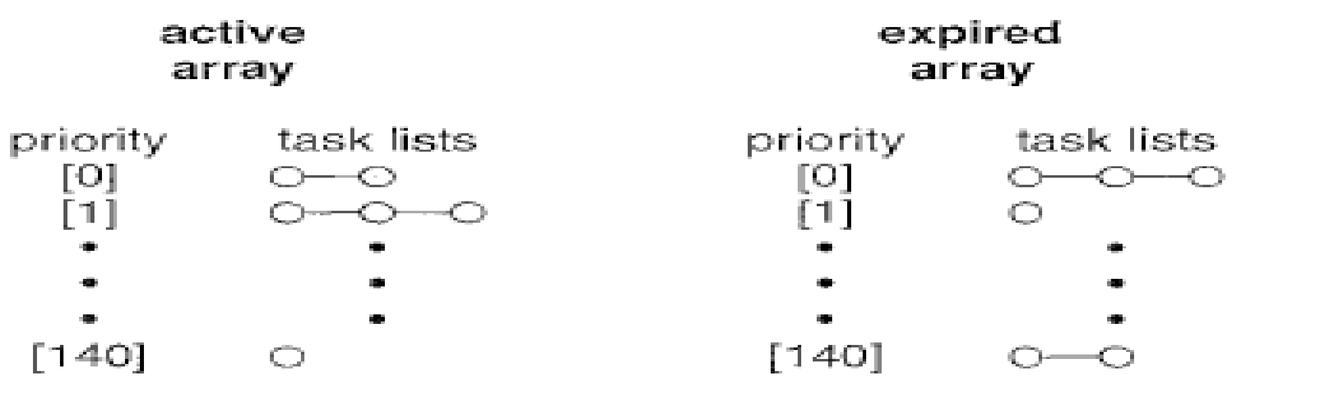


Figure 5.16 List of tasks indexed according to priority.

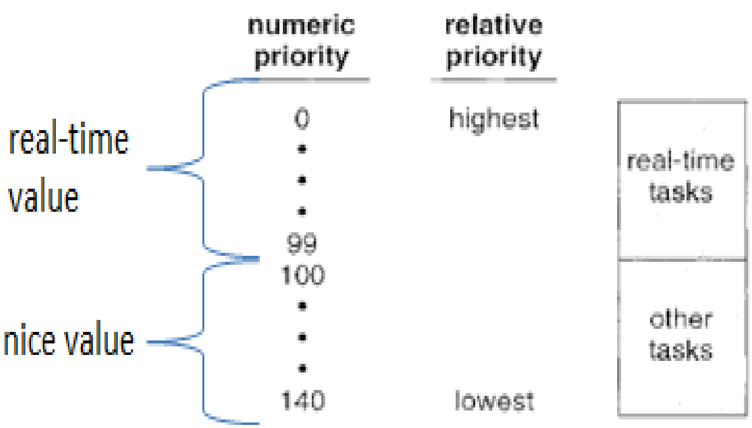
Linux Scheduling Through Version 2.5

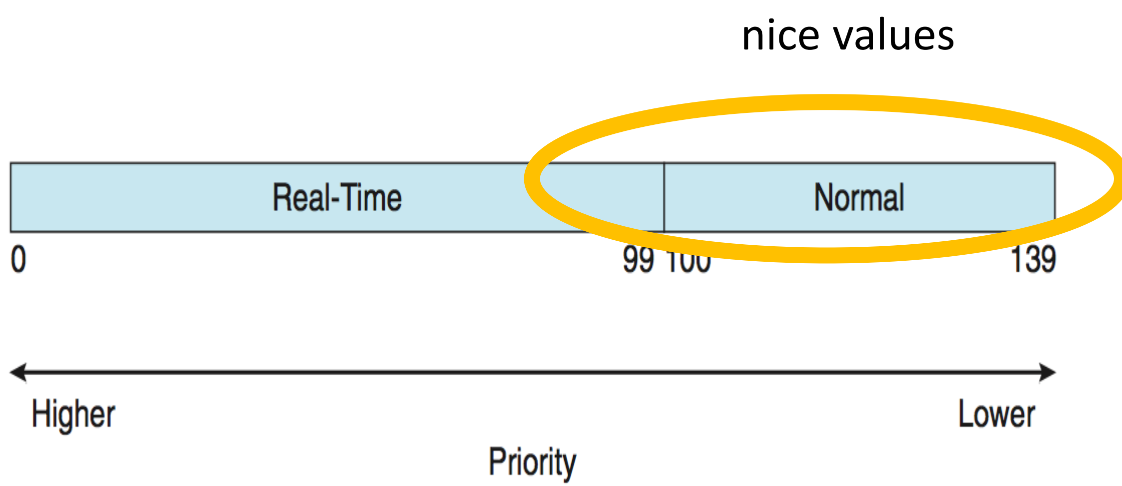
- On multiprocessor machines,
  - each processor is scheduling the highest-priority task
  - from its own runqueue structure.

- When all tasks have exhausted their time slices (that is, the active array is empty),
  - **the two priority arrays are exchanged;**
  - **the expired array becomes the active array, and vice versa.**

Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - **Real-time tasks are assigned static priorities**
- **Other tasks have**
  - **dynamic priorities** that are based on their *nice* values plus or minus the value 5.
  - The interactivity of a task determines whether
    - the value 5 will be added to or subtracted from the *nice* value.



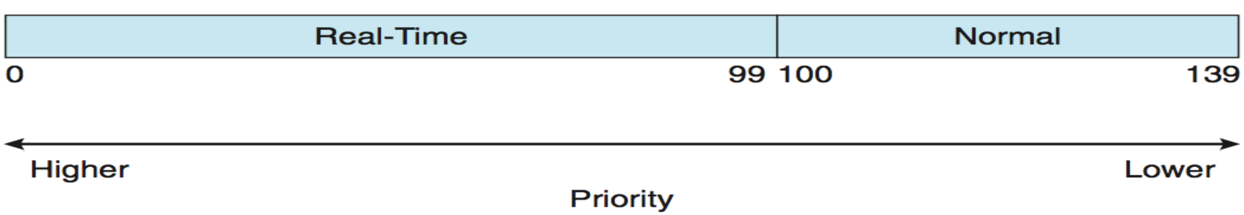


Linux Scheduling (Cont.)

- A task's interactivity is determined by
  - how long it has been sleeping while waiting for I/0.

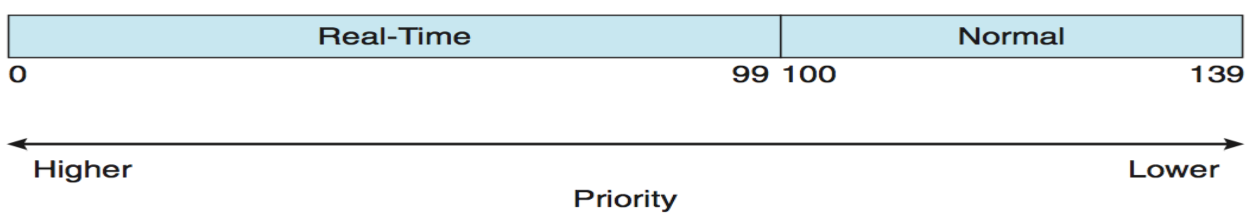
Linux Scheduling (Cont.)

- **Tasks that are more interactive typically have longer sleep times and therefore are more likely to have adjustments closer to -5,**
- As the scheduler favors interactive tasks, The result of such adjustments will be higher priorities for these tasks.



Linux Scheduling (Cont.)

- Conversely, tasks with shorter sleep times are often more CPU-bound and thus will have their priorities lowered.



Linux Scheduling (Cont.)

- Interactive Processes ?
  - **These interact constantly with their users, and therefore spend a lot of time waiting for keypresses and mouse operations.**
  - **When input is received, the process must be woken up quickly, or the user will find the system to be unresponsive.**
  - Typically, the average delay must fall between 50 and 150 ms.
  - The variance of such delay must also be bounded, or the user will find the system to be erratic.
  - Typical interactive programs are command shells, text editors, and graphical applications.

Linux Scheduling in Version 2.6.23 +

Completely Fair Scheduler (CFS)

- It is default scheduling process since version 2.6.23.
- Elegant handling of I/O and CPU bound process.
- **Each runnable process have a virtual time associated with it in PCB (process control block).**

Linux Scheduling in Version 2.6.23 +

- Whenever a context switch happens
  - **then current running process virtual time is increased by**
  - **virtualruntime_currprocess+=T.**
    where T is time for which it is executed recently.

- Runtime for the process therefore monotonically **increases.**

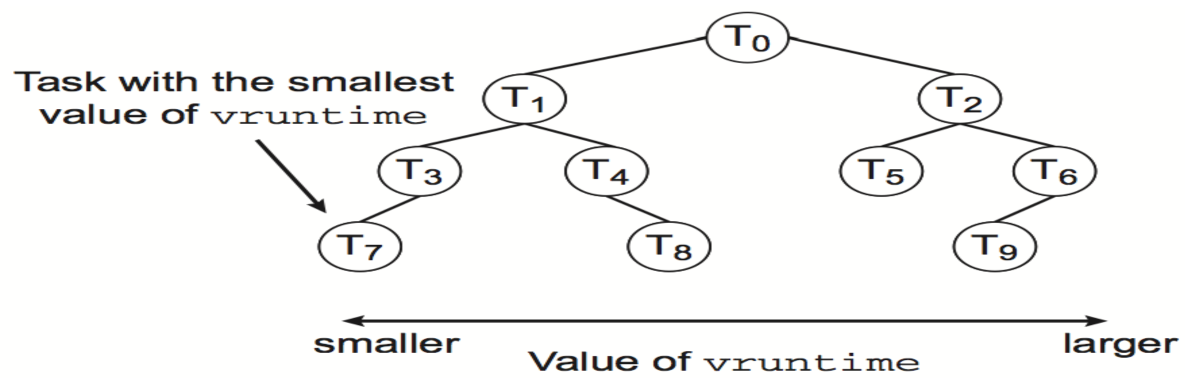
- **So initially every process have some starting virtual time**

Linux Scheduling in Version 2.6.23 +

- CFS is quite simple algorithm for the process scheduling
- It is implemented **using RED BLACK Trees and not queues.**
  - **So all the process which are on main memory are inserted into Red Black trees and**
  - whenever a new process comes it is inserted into the tree.
  - As we know that Red Black trees are self Balancing binary Search trees.

Linux Scheduling in Version 2.6.23 +

- When **context switch** occurs –
  - The virtual time for the current process which was executing is updated .
  - The new process is decided
    - which has lowest virtual time and
    - that we know that is left most node of Red Black tree.



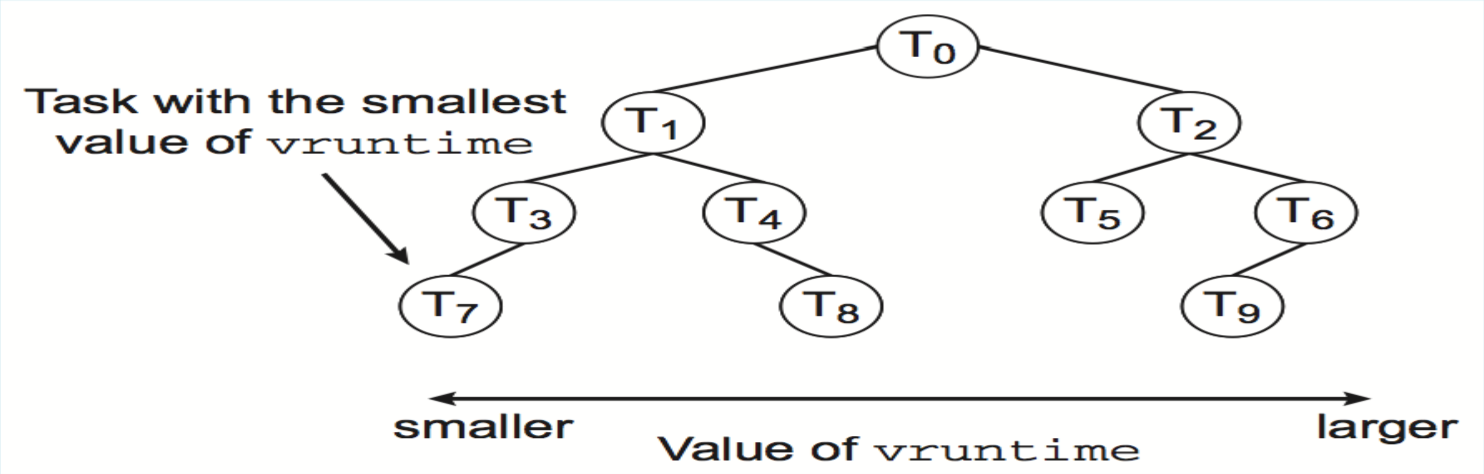
Linux Scheduling in Version 2.6.23 +

- If the current process still has some burst time then it is inserted into the Red Black tree.

- So this way each process gets fair time for the execution as
  - after every context switch the virtual time of a process increases and
  - thus priority shuffles.

CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Thread Scheduling

Thread Scheduling

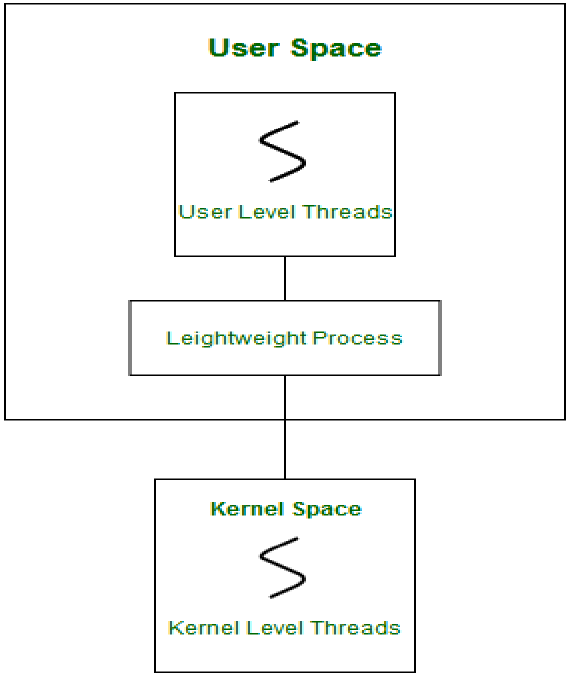
- Distinction between user-level and kernel-level threads

- On operating systems that support them,
  - **it is kernel-level threads-not processes-that are being scheduled by the operating system.**

  - **User-level threads are managed by a thread library, and the kernel is unaware of them.**

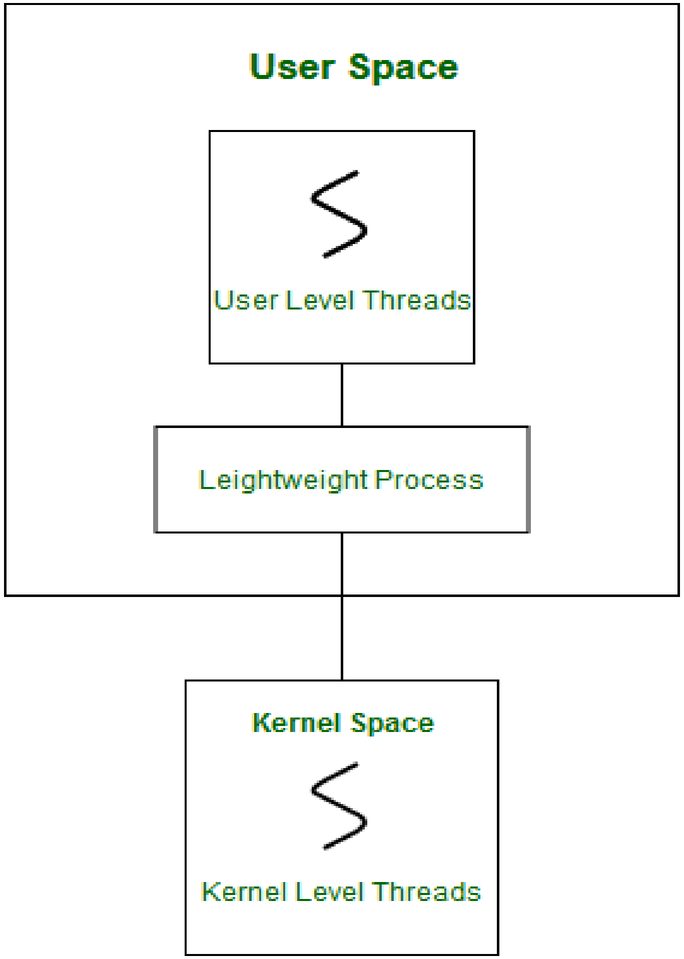
Thread Scheduling

- To run on a CPU, user-level threads
  - **must ultimately be mapped to an associated kernel-level thread,**
  - Although this mapping may be indirect and may use a lightweight process (LWP).



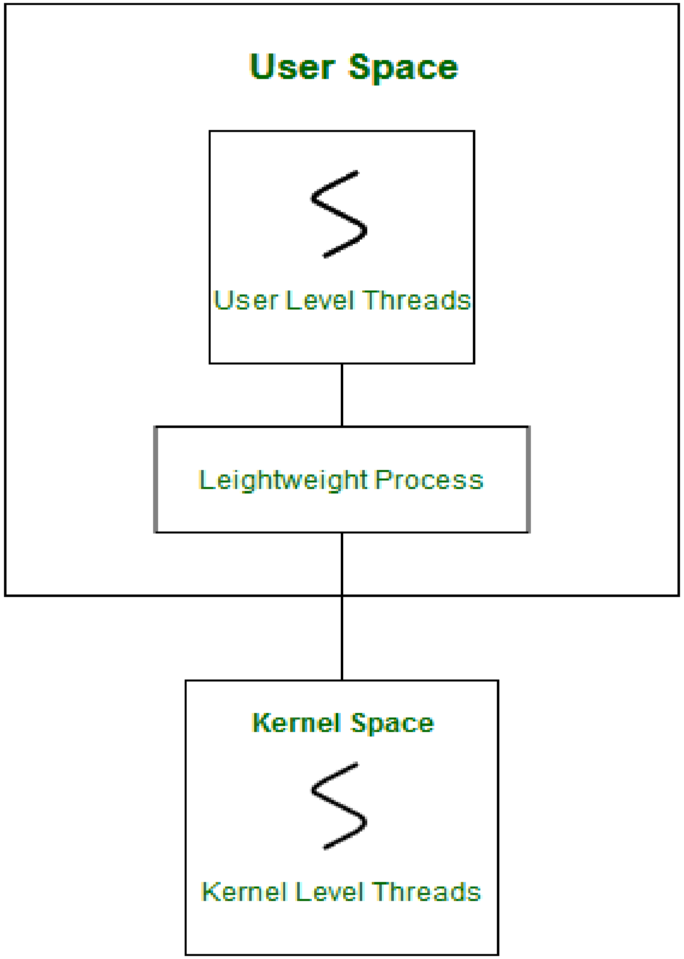
Thread Scheduling

- 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 Scheduling

- LWP-
  - Thread library schedules which thread of a process to run on which LWP and how long.
  - The number of LWP created by the thread library depends on the type of application



Thread Scheduling

- When we say the thread library *schedules* user threads onto available LWPs,
  - **we do not mean that the thread is actually running on a CPU;**
  - **this would require the operating system to schedule the kernel thread onto a physical CPU.**

Process Contention Scope (PCS)

- Many-to-one and many-to-many models,
  - **thread library schedules user-level threads to run on LWP**
- **Known as process-contention scope (PCS) since scheduling competition is within the process**
- **Typically done via priority set by programmer**

- The contention takes place among threads **within a same process**.

System Contention Scope (SCS)

- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)**
- **i.e. Decide which Kernel thread to execute on the CPU**
- **Competition for the CPU with SCS scheduling takes place among all kernel threads in system**

- The contention takes place among **all kernel threads in the system**

PCS Scheduling

- PCS is done according to priority-
  - the scheduler selects the runnable thread with the highest priority to run.

- User-level thread priorities
  - **are set by the programmer and**
  - **are not adjusted by the thread library,**
  - **Although some thread libraries may allow the programmer to change the priority of a thread.**

PCS Scheduling

PCS will typically

- **preempt the thread currently running in favor of a higher-priority thread;**
- However, there is no guarantee of time slicing among threads of equal priority.

Pthread

- **POSIX Threads,**
- **Referred to as pthreads,**
- An execution model that exists independently from a language,
- As a parallel execution model.
- **POSIX Threads is an API defined by the standard POSIX.1c**

Pthread

- It allows a program to control multiple different flows of work that overlap in time.
- Each flow of work is referred to as a thread.
- Creation and control over these flows is achieved by making calls to the POSIX Threads API.

Pthread Scheduling

- API allows specifying –
- **Either PCS or SCS during thread creation**
  - **PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling**
  - **PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling**

Pthread Scheduling

- Systems using the one-to-one model
  - such as Windows XP, Solaris, and Linux,
  - schedule threads using only SCS.

- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling

- The Pthread IPC provides two functions **for getting-and setting-the contention scope policy:**
  - **pthread_attr_setscope(pthread_attr_t *attr, int scope)**
  - **pthread_attr_getscope(pthread_attr_t *attr, int *scope)**

Pthread Scheduling

pthread_attr_setscope(pthread_attr_t *attr, int scope)

Description:

- Sets the thread contention scope attribute in the thread attribute object attr to the value specified in scope.

- Sets the scope attribute in the **attr** object.

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fp%2Fpthread_attr_setscope.html

Pthread Scheduling

pthread_attr_setscope(pthread_attr_t *attr, int scope)

- 1st parameter for both functions
  - attr -Specifies the thread attributes object.
  - A pointer to the pthread_attr_t structure that defines the attributes to use when creating new threads

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fp%2Fpthread_attr_setscope.html

Pthread Scheduling

pthread_attr_setscope(pthread_attr_t *attr, int scope)

- 2nd parameter :
  - The new value for the contention scope attribute
  - The second parameter defines the scope of contention for the thread pointed.
  - It takes two values.
  - PTHREAD_SCOPE_SYSTEM
  - PTHREAD_SCOPE_PROCESS
  - indicating how the contention scope is to be set.

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fp%2Fpthread_attr_setscope.html

Pthread Scheduling

pthread_attr_setscope(pthread_attr_t *attr, int scope)

Return Values-

- Upon successful completion, the pthread_attr_getscope and pthread_attr_setscope subroutines return a value of 0.
- Otherwise, an error number is returned to indicate the error.

https://www.ibm.com/docs/en/aix/7.2?topic=p-pthread-attr-getscope-pthread-attr-setscope-subroutines

Pthread Scheduling

pthread_attr_getscope(pthread_attr_t *attr, int *scope)

Description:

- Gets the thread contention scope attribute from the thread attribute object *attr* and returns it in *scope*.

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fp%2Fpthread_attr_setscope.html

Pthread Scheduling

pthread_attr_getscope(pthread_attr_t *attr, int *scope)

- 1st parameter for both functions
  - attr -Specifies the thread attributes object.
  - A pointer to the pthread_attr_t structure that defines the attributes to use when creating new threads

- 2nd parameter :
  - A pointer to a location where the function can store the current contention scope.

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fp%2Fpthread_attr_setscope.html

pthread_attr_t *attr

- **The thread attributes structure**
- When you start a new thread, it can assume some well-defined defaults, or you can explicitly specify its characteristics.

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.getting_started%2Ftopic%2Fs1_procs_thread_attr.html

pthread_attr_t *attr

- Let's look at the pthread_attr_t data type:

```
typedef struct {
    int                 __flags;
    size_t              __stacksize;
    void                *__stackaddr;
    void                (*__exitfunc)(void *status);
    int                 __policy;
    struct sched_param  __param;
    unsigned            __guardsize;
} pthread_attr_t;
```

pthread_attr_t *attr

```
typedef struct {
    int                     __flags;
    size_t                  __stacksize;
    void                    *__stackaddr;
    void                    (*__exitfunc)(void *status);
    int                     __policy;
    struct sched_param      __param;
    unsigned                __guardsize;
} pthread_attr_t;
```

Basically, the fields are used as follows:

- __flags -Non-numerical (Boolean) characteristics
- __stacksize, __stackaddr, and __guardsize-Stack specifications.
- __exitfunc- Function to execute at thread exit.
- __policy and __param- Scheduling parameters.

Pthread Example

- The program first determines the existing contention scope and sets it to PTHREAD_SCOPE_SYSTEM.

- It then creates five separate threads that will run using the SCS scheduling policy.

Pthread Scheduling

pthread_attr_init()

- *Initialize a thread-attribute object*

Synopsis:

#include <pthread.h>

int pthread_attr_init(pthread_attr_t **attr*);

Arguments:

- *attr* -A pointer to the pthread_attr_t structure that you want to initialize.

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fp%2Fpthread_attr_setscope.html

Pthread Example

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API

```
    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i],&attr,runner,NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

Pthread Scheduling

pthread_create()

- *Create a thread*

Synopsis:

#include <pthread.h>

int pthread_create(pthread_t* *thread*, const pthread_attr_t* *attr*, void* (**start_routine*)(void*), void* *arg*);

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fp%2Fpthread_attr_setscope.html

Pthread Scheduling

int pthread_create(pthread_t* *thread*, const pthread_attr_t* *attr*, void* (**start_routine*)(void*), void* *arg*);

Arguments:

- *thread* –
  - a pointer to a pthread_t object where the function can store the thread ID of the new thread.

pthread_t

- pthread_t is the data type used to uniquely identify a thread.
- Used by the application in function calls that require a thread identifier.

Pthread Scheduling

int pthread_create(pthread_t* *thread*, const pthread_attr_t* *attr*, void* (**start_routine*)(void*), void* *arg*);

Arguments:

- *attr* -
  - A pointer to a pthread_attr_t structure that specifies the attributes of the new thread.
  - Instead of manipulating the members of this structure directly, use pthread_attr_init() and the pthread_attr_set_* functions.
  - If you modify the attributes in *attr* after creating the thread, the thread's attributes aren't affected.

Pthread Scheduling

int pthread_create(pthread_t* *thread*, const pthread_attr_t* *attr*, void* (**start_routine*)(void*), void* *arg*);

Arguments:

- *start_routine*-
  - The routine where the thread begins, with *arg* as its only argument.

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fp%2Fpthread_attr_setscope.html

Pthread Scheduling

int pthread_create(pthread_t* *thread*, const pthread_attr_t* *attr*, void* (**start_routine*)(void*), void* *arg*);

Arguments:

- *arg* The argument to pass to *start_routine*.

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fp%2Fpthread_attr_setscope.html

pthread_create()

int pthread_create(pthread_t ****thread*, const pthread_attr_t**
******attr***, void *(****start_routine***) (void *), void ****arg***);**

RETURN VALUE

- On success, **pthread_create**() returns 0;
- On error, it returns an error number, and the contents of **thread* are undefined.

Pthread Scheduling

- int pthread_create(pthread_t* *thread*, const pthread_attr_t* *attr*, void* (**start_routine*)(void*), void* *arg*);

- `pthread_create(&tid[i],&attr,runner,NULL);`

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fp%2Fpthread_attr_setscope.html

Pthread Scheduling API

```
    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i],&attr,runner,NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

Pthread Scheduling

Joining

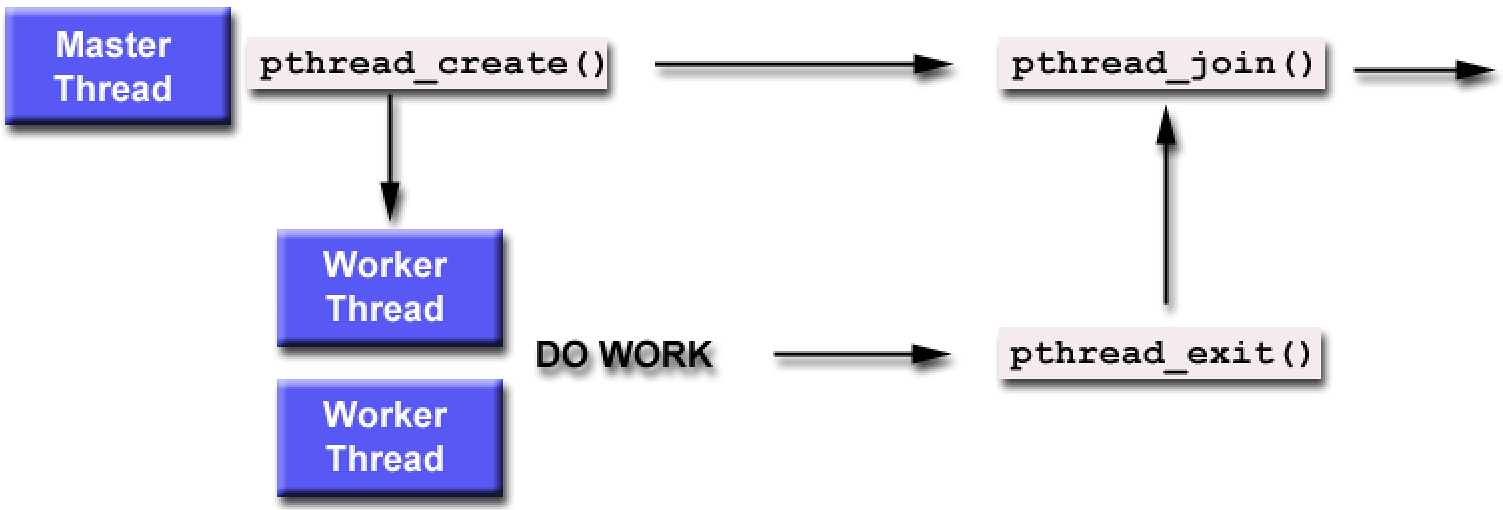
- The simplest method of synchronization is to *join* the threads as they terminate.

- Joining really means waiting for termination.

- Joining is accomplished by one thread waiting for the termination of another thread. The waiting thread calls pthread_join()

Pthread Scheduling

Joining



Pthread Scheduling

int pthread_join(pthread_t *thread*, void** *value_ptr*);

`Eg-pthread_join(tid[i], NULL);`

Arguments:

- *thread* –
  - The target thread whose termination you're waiting for,
  - Pass it the thread ID of the thread that you wish to join

- *value_ptr* –
  - An optional *value_ptr*, which can be used to store the termination return value from the joined thread.
  - NULL, or a pointer to a location where the function can store the value passed to pthread_exit() by the target thread.
  - You can pass in a NULL if you aren't interested in this value

Pthread Scheduling

- The pthread_join function returns an integer value that also indicates different error codes.

Return Value-

- 0 if the call was successful and this guarantees the given thread has terminated.
- EDEADLK- a deadlock was detected.
- EINVAL- the given thread is not joinable
- ESRCH- the given thread ID can’t be found.

Pthread Scheduling API

```
    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

The calling thread waits for every thread with the pthread_join call in the loop

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope)
!= 0)
      fprintf(stderr, "Unable to get
scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope ==
PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope
value.\n");
   }
```

```
/* set the scheduling algorithm to PCS
or SCS */
   pthread_attr_setscope(&attr,
PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
pthread_create(&tid[i],&attr,runner,NUL
L);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in
this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Pthread Scheduling

pthread_exit()

- *Terminate a thread*

Synopsis:

#include <pthread.h>

void pthread_exit(void* *value_ptr*);

Pthread Scheduling

void pthread_exit(void* *value_ptr*);

Arguments:

- *value_ptr* -A pointer to a value that you want to be made available to any thread joining the thread that you're terminating.

Description:

- Terminates the calling thread.
- This routine kills the thread
- If the thread is joinable, the value *value_ptr* is made available to any thread joining the terminating thread (only one thread can get the return status).
- If the thread is detached, all system resources allocated to the thread are immediately reclaimed.

Diff between exit and joining?

- pthread_exit is called from the thread itself to terminate its execution (and return a result) early.

- pthread_join is called from another thread (usually the thread that created it) to wait for a thread to terminate and obtain its return value.
- It can be called before or after the thread you're waiting for calls pthread_exit.
- If before, it will wait for the exit to occur.
- If after, it simply obtains the return value and releases the pthread_t resources.

https://codehunter.cc/a/c/difference-between-pthread-exit-pthread-join-and-pthread-detach

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep()
#include <pthread.h>

// A normal C function that is executed as a
thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
   sleep(1);
   printf("Printing Hello from Thread \n");
   return NULL;
}

int main()
{
   pthread_t thread_id;
   printf("Before Thread\n");
   pthread_create(&thread_id, NULL,
myThreadFun, NULL);
   pthread_join(thread_id, NULL);
   printf("After Thread\n");
   exit(0);
}
```

Output ?

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep()
#include <pthread.h>

// A normal C function that is executed as a
thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Printing Hello from Thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL,
myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

```
Output-


Before Thread
Printing Hello from Thread
After Thread
```

main

```
#include <iostream>
#include <stdlib>
#include <pthread.h>
using namespace std;
#define NUM_THREADS 5
void *PrintHello(void *threadid) {
  long tid;
  tid = (long)threadid;
  printf("Hello World! Thread ID, %d\n", tid);
  pthread_exit(NULL);
}
```

```
int main () {
  pthread_t threads[NUM_THREADS];
  int rc;
  int i;
  for( i = 0; i < NUM_THREADS; i++ ) {
    cout << "main() : creating thread, " << i << endl;
    rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);
    if (rc) {
      printf("Error:unable to create thread, %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

Output ?

```cpp
#include <iostream>
#include <stdlib>
#include <pthread.h>
using namespace std;
#define NUM_THREADS 5
void *PrintHello(void *threadid) {
  long tid;
  tid = (long)threadid;
  printf("Hello World! Thread ID, %d\n", tid);
  pthread_exit(NULL);
}
int main () {
  pthread_t threads[NUM_THREADS];
  int rc;
  int i;
  for( i = 0; i < NUM_THREADS; i++ ) {
    cout << "main() : creating thread, " << i << endl;
    rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);
    if (rc) {
      printf("Error:unable to create thread, %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

IDEAL?

```
Output-


main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 0
Hello World! Thread ID, 1
Hello World! Thread ID, 2
Hello World! Thread ID, 3
Hello World! Thread ID, 4
```

Sequence/No Sequence?

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 0
Hello World! Thread ID, 2
Hello World! Thread ID, 1
Hello World! Thread ID, 3
Hello World! Thread ID, 4
```

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 4
Hello World! Thread ID, 3
Hello World! Thread ID, 0
Hello World! Thread ID, 1
Hello World! Thread ID, 2


...Program finished with exit code 0
```

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 2
Hello World! Thread ID, 4
Hello World! Thread ID, 1
Hello World! Thread ID, 0
Hello World! Thread ID, 3


...Program finished with exit code 0
```

Playing around with the Code

main.cpp

```cpp
#include<stdlib.h>
#include<pthread.h>
using namespace std;
#define NUM_THREADS 5
void *PrintHello(void *threadid) {
    long tid;
    tid = (long)threadid;
    printf("Hello World! Thread ID, %d\n", tid);
    printf("Before termination of Thread ID, %d\n", tid);
    pthread_exit(NULL);
    printf("After termination of Thread ID, %d\n", tid);
}
int main () {
    pthread_t threads[NUM_THREADS];
    int rc;
    int i;
    for( i = 0; i < NUM_THREADS; i++ ) {
        cout << "main() : creating thread, " << i << endl;
        rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);
        if (rc) {
            printf("Error:unable to create thread, %d\n", rc);
            exit(-1);
        }
    }
     printf("Before termination of Main Thread\n");
    pthread_exit(NULL);
    printf("After termination of Main Thread\n");
}
```

Playing around with the Code

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
Hello World! Thread ID, 0
Before termination of Thread ID, 0
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 1
Before termination of Thread ID, 1
Before termination of Main Thread
Hello World! Thread ID, 3
Before termination of Thread ID, 3
Hello World! Thread ID, 4
Before termination of Thread ID, 4
Hello World! Thread ID, 2
Before termination of Thread ID, 2


...Program finished with exit code 0
Press ENTER to exit console.
```