Chapter 3: CPU Scheduling

Mr. SUBHASIS DASH

SCHOLE OF COMPUTER ENGINEERING.

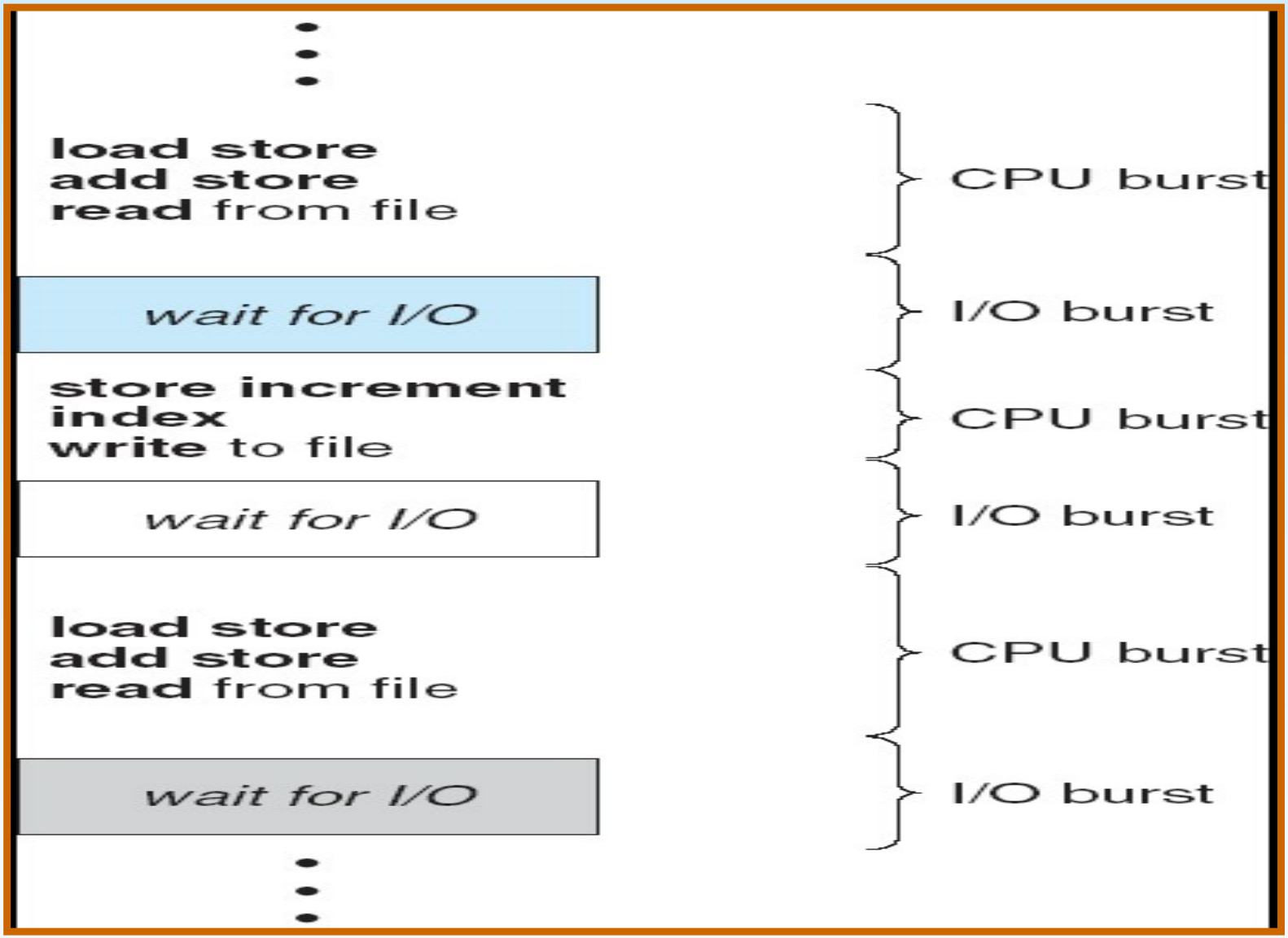
KIIT UNIVERSITY

BHUBANESWAR

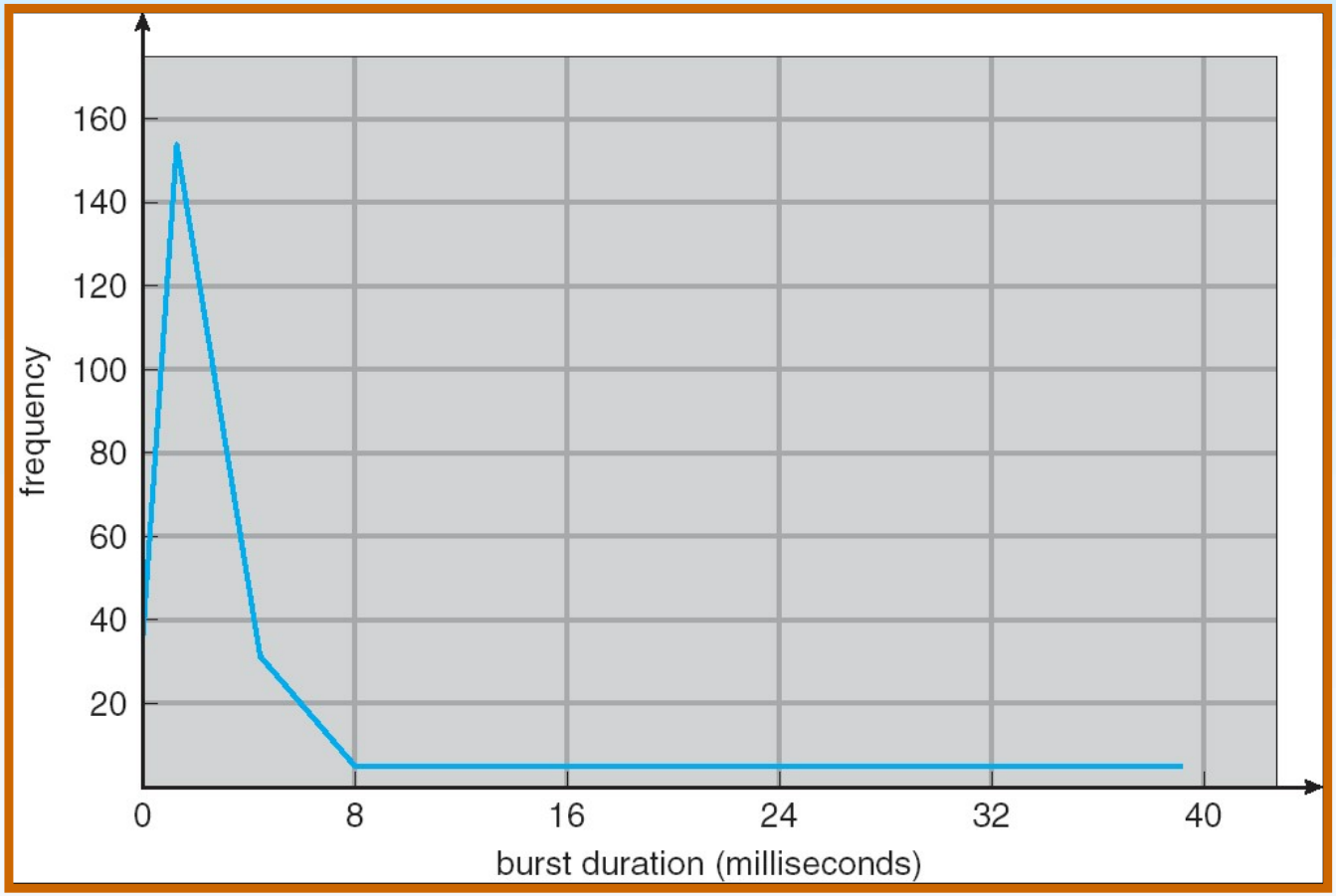
Basic Concepts

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

Alternating Sequence of CPU And I/O Bursts



Histogram of CPU-burst Times



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. **Switches from running to waiting state**
  2. **Switches from running to ready state**
  3. **Switches from waiting to ready**
  4. **Terminates**
- Scheduling under 1 and 4 is *non-preemptive*
- All other scheduling is *preemptive*

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

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

Optimization Criteria

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

First-Come, First-Served (FCFS) Scheduling

- Process request the CPU first also allocated with the CPU first [FIFO queue].
- Here the average waiting time is quit long.

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:

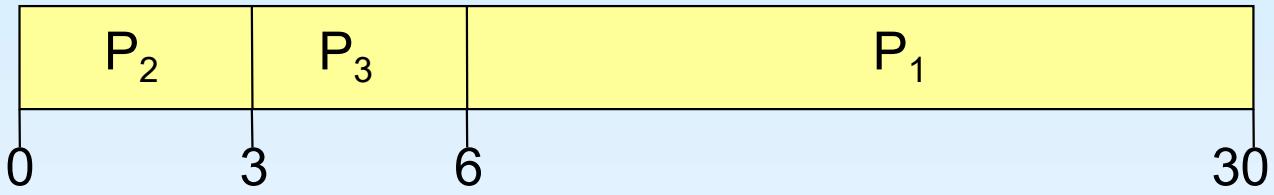
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 51/3=17
- Average turnaround time = (30 + 51) /3 = 27

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Average turnaround time = (30 + 9) /3 = 13
- Much better than previous case
- Here average wait time is dependant upon the situation of the process arrives at the ready queue.
- *Convoy effect* -short process behind long process

CPU bound & I/O bound

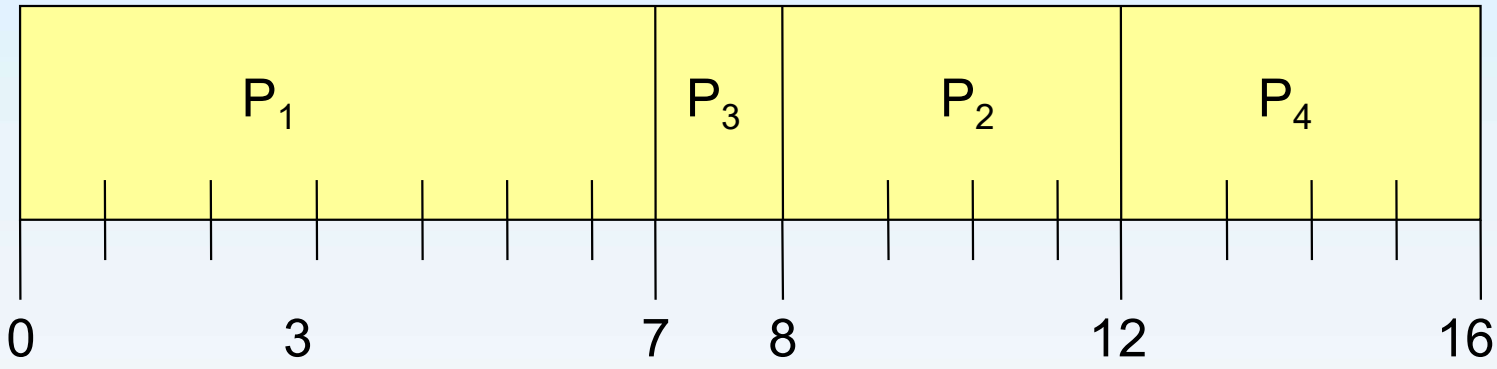
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst.
- Use these lengths to schedule the process with the shortest time
- If two processes have same CPU burst time, then FCFS is applied
- Two schemes:
  - **Non-preemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst
  - **Preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

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

- SJF (non-preemptive)

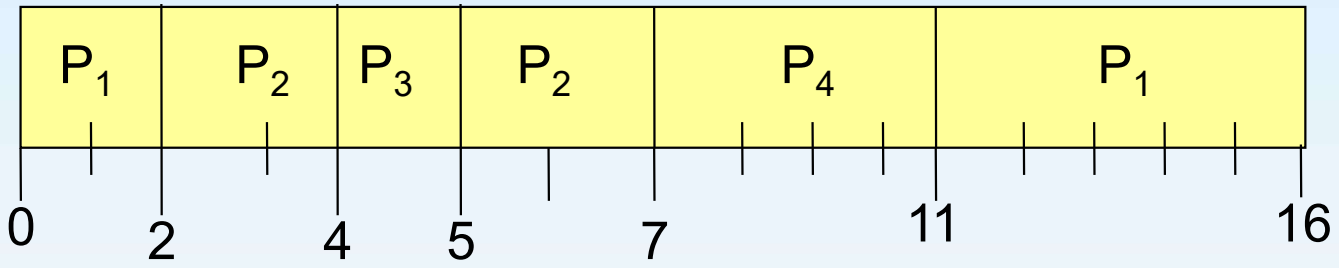


- Wait time for $P_1 = 0$, $P_2 = 8 - 2 = 6$, $P_3 = 7 - 4 = 3$, $P_4 = 12 - 5 = 7$
- Average waiting time = (0 + 6 + 3 + 7)/4 = 4
- Average turnaround time = (16 + 16) /4 = 8

Example of Preemptive SJF

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

- SJF (preemptive)



- Wait time for $P_1 = 0 + (11 – 2) = 9$, $P_2 = (2 - 2) + (5 – 4) = 1$, $P_3 = 4 - 4 = 0$, $P_4 = 7 - 5 = 2$
- Average waiting time = (9 + 1 + 0 +2)/4 = 3
- Average turnaround time = (16 + 12) /4=7
- Better than the non-preemptive example.

DIFFICULTIES WITH SJF ALGORITHM

- To know the length of the next process CPU burst which is shortest one.
- Estimate the next shortest CPU burst
- SJF can be used for long-term but can’t for short-term scheduler.

Determining Length of Next CPU Burst

- Can only estimate the length
- Next CPU burst length may be predicted.
- Can be done by using the length of previous CPU bursts, using exponential averaging
- Start with an exception :-

Next CPU burst = length of the previous CPU burst

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

Examples of Exponential Averaging

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

- Where τ_n = stores the pervious CPU burst time

  α = Parameter controls the relative weight of the recent & past history in our prediction.

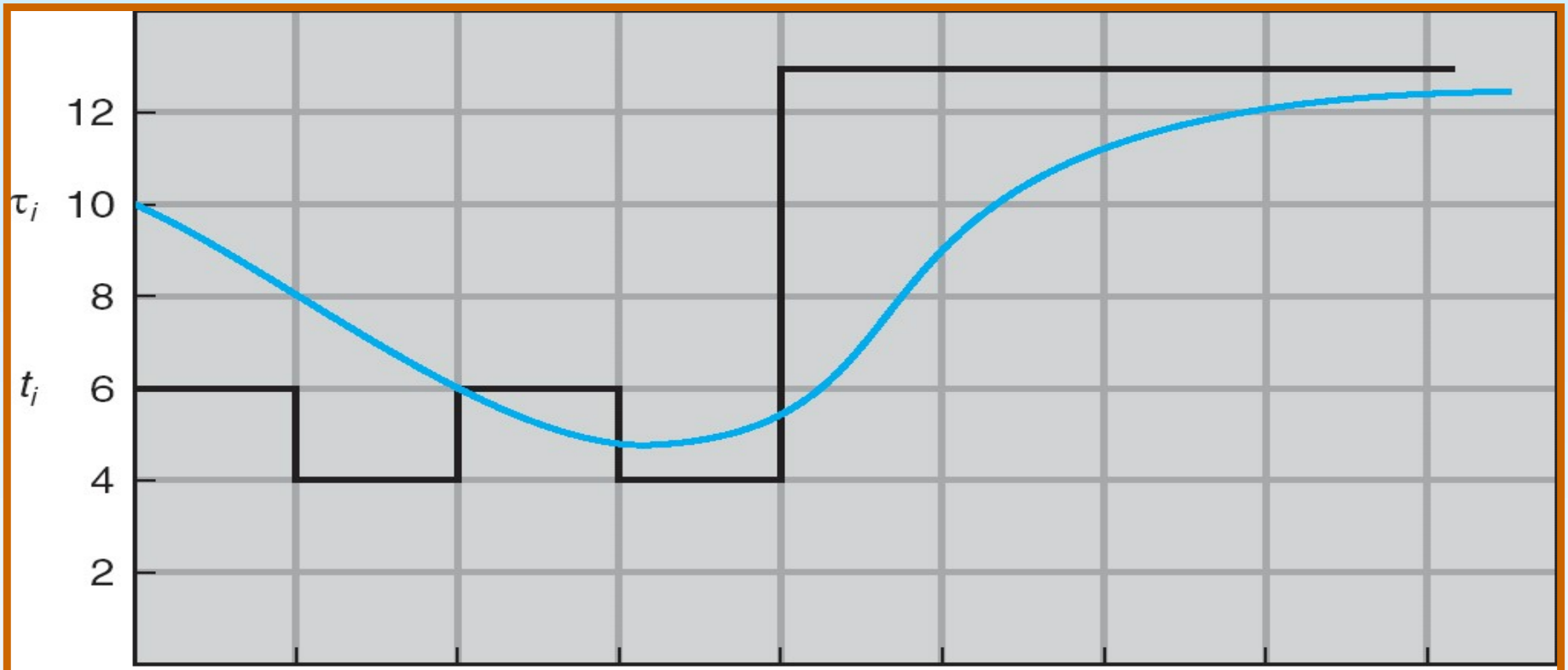
- If $\alpha = 0$, then $\tau_{n+1} = \tau_n$ (Recent history does not have any effect)
- If $\alpha = 1$, then $\tau_{n+1} = \alpha\, t_n$ (pervious CPU burst does not have nay effect)
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + (1- \alpha)^2\, \alpha\, t_{n-2} +$$

$$\ldots. + \quad (1 - \alpha)^j\, \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1}\, \tau_0$$

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

Prediction of the Length of the Next CPU Burst



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

Priority Scheduling

- SJF is also one type of priority algorithm.
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- A priority number (integer) is associated with each process
- Process having higher CPU burst time = Lower the priority
- Process having lower CPU burst time = Higher the priority
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
- A Priority can be defined either **internally** or **externally**.
- If two same priority appears then apply SJF (if same arrival time then FCFS).
- Two types of schemes
  - **Preemptive**
  - **Non-preemptive**

Example of Non-Preemptive Priority

- The given data:

Process	Arrival time	Burst time	Priority
P_1	00	10	8
P_2	03	02	1
P_3	05	07	5
P_4	06	05	3
P_5	08	08	6

- SJF (non-preemptive)

- Wait time for $P_1 = 0$, $P_2 = 10 - 3 = 7$, $P_3 = 17 - 5 = 12$

  $P_4 = 12 - 6 = 6$, $P_5 = 24 - 8 = 16$

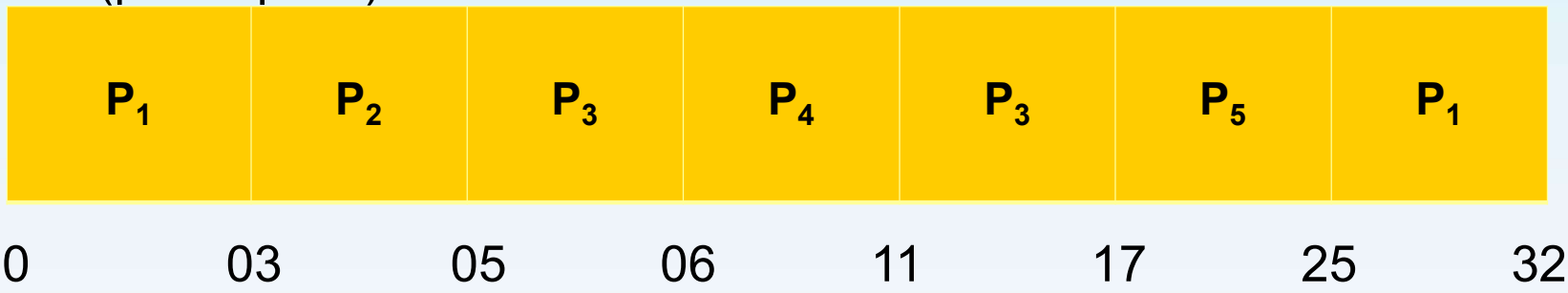
- Average wait time = $(0 + 7 + 12 + 6 + 16) / 5 = 8.2$

Example of Preemptive Priority

- The given data:

Process	Arrival time	Burst time	Priority
P_1	00	10	8
P_2	03	02	1
P_3	05	07	5
P_4	06	05	3
P_5	08	08	6

- SJF (preemptive)



- Wait time for $P_1 = 0 + (25 – 3) = 22$, $P_2 = 3 – 3 = 0$, $P_3 = (5 – 5) + (11- 6) = 5$

  $P_4 = 6 – 6 = 0$, $P_5 = 17 – 8 = 09$

- Average wait time = (22 + 0 + 5 + 0 + 09) / 5 = 7.2
- ***Better than the non-preemptive example.***

Priority Scheduling

- Conclusion :-
  1. Non-preemptive just add new process to the head of the queue.
  2. Preemptive compares the new process with currently executing process and allow the process with higher priority.

- **Problem** ≡ STARVATION → Low priority processes may never execute

- **Solution** ≡ AGING →as time progresses increase the priority of the process

Round Robin (RR) Scheduling

- Especially for time sharing system.
- FCFS + Preemptive execution ≡ Round Robin
- Each process gets a small unit of CPU time (*time quantum*).
- Time Quantum usually 10-100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If time quantum > CPU burst time for a process then no preemption.
- If time quantum < CPU burst time for a process then preempt the process.
- Performance a quantum time :-

  Quantum Time very large ≡ FIFO

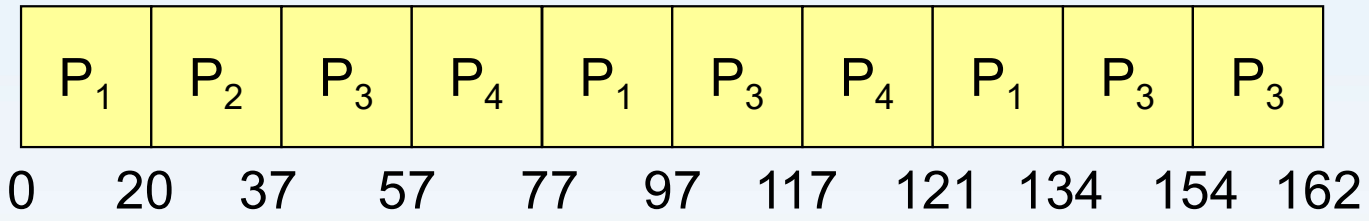
  Quantum Time very small ≡ Round Robin will act as process sharing.

  Quantum time must be large with respect to context switch, otherwise overhead is too high .
- 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.

Example of RR with Time Quantum = 20

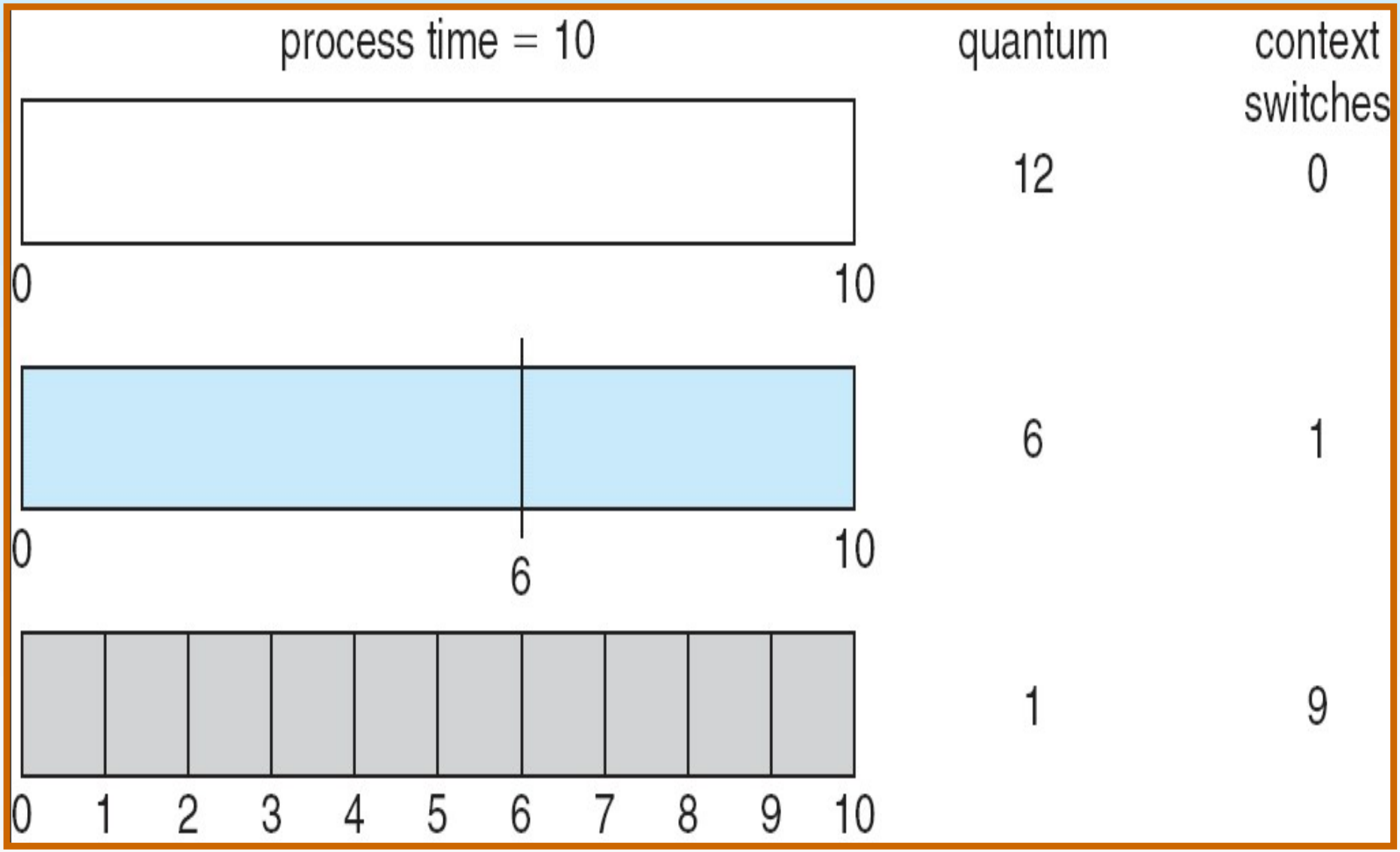
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

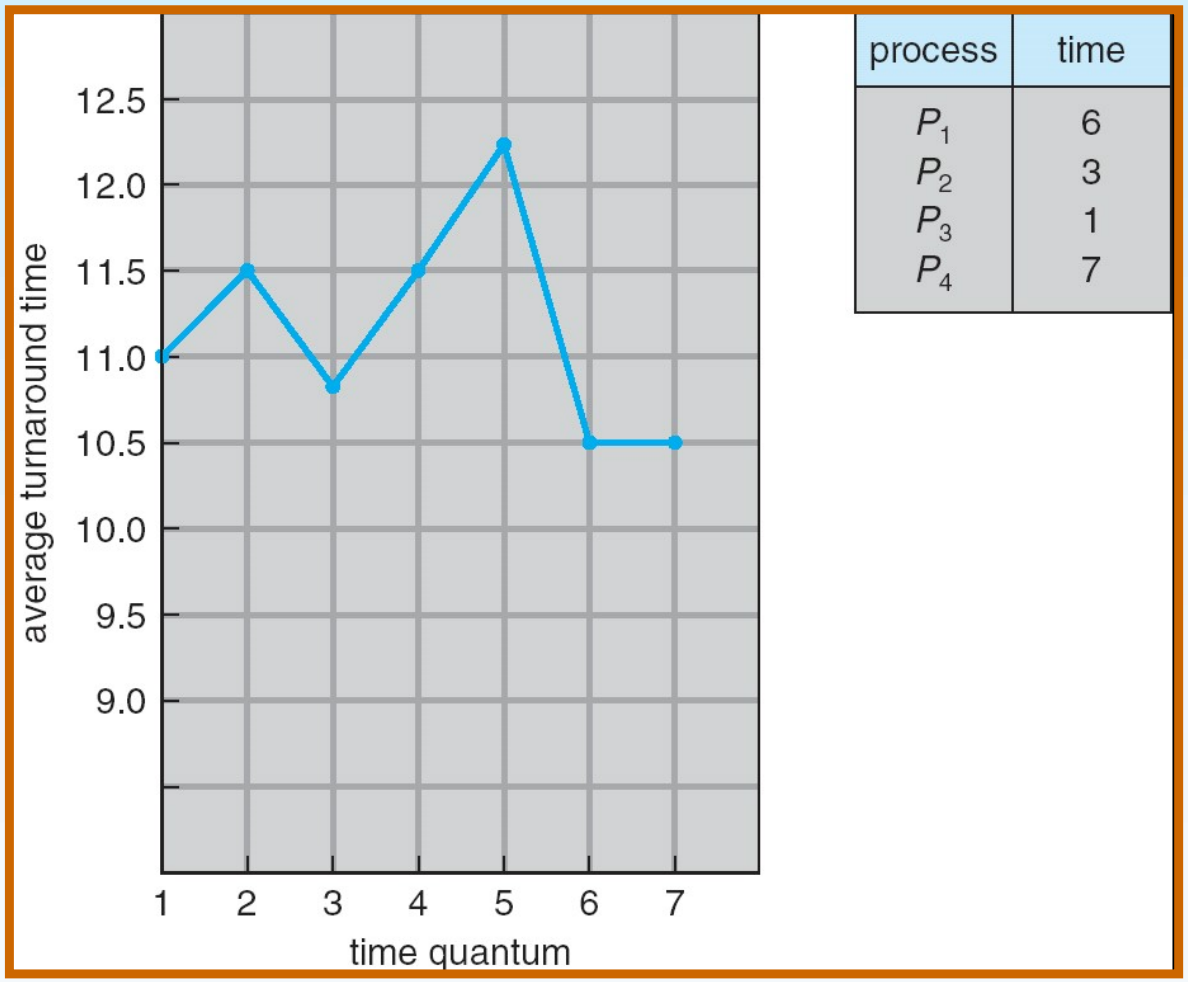


- Typically, higher average turnaround than SJF, but better *response*

Time Quantum & Context Switch Time



Turnaround Time Varies With The Time Quantum

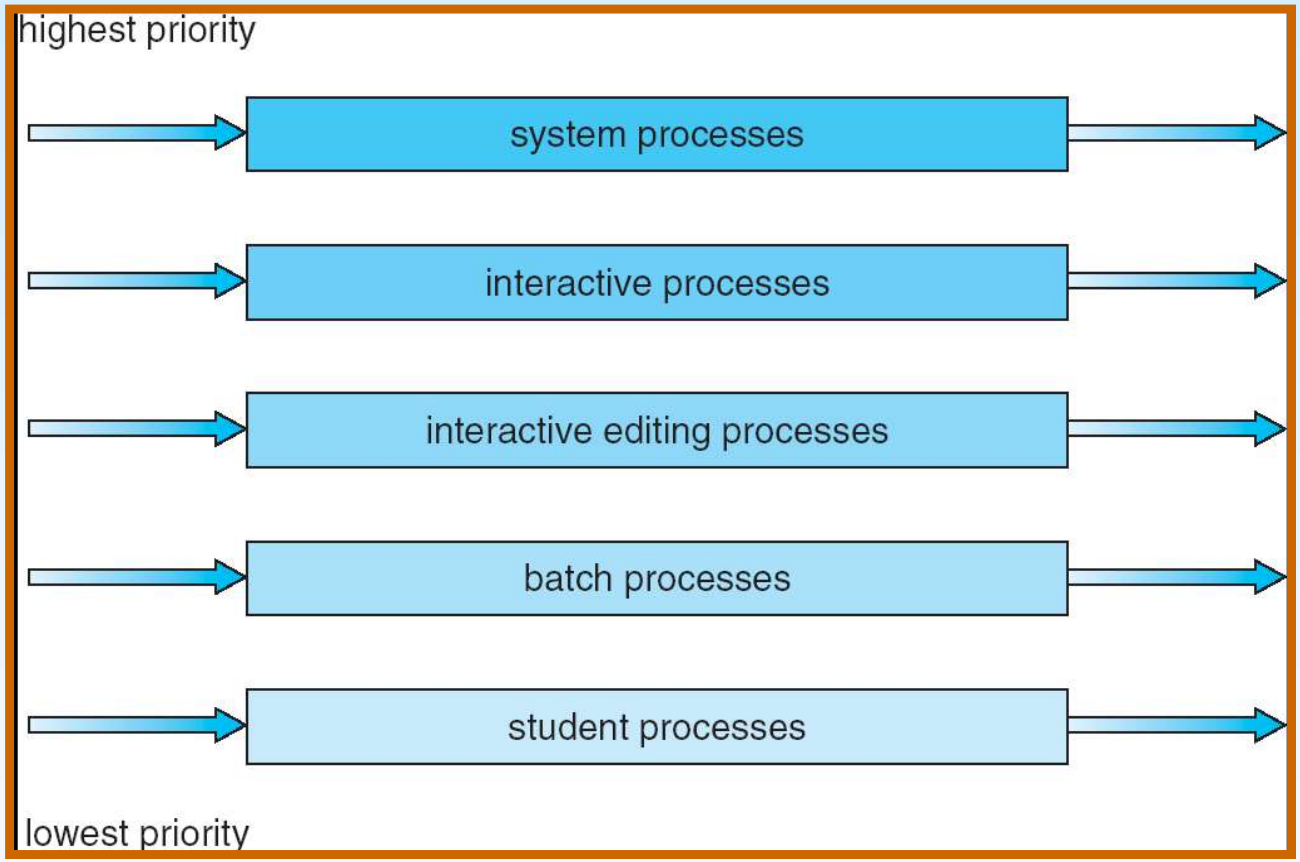


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

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

Multilevel Queue Scheduling



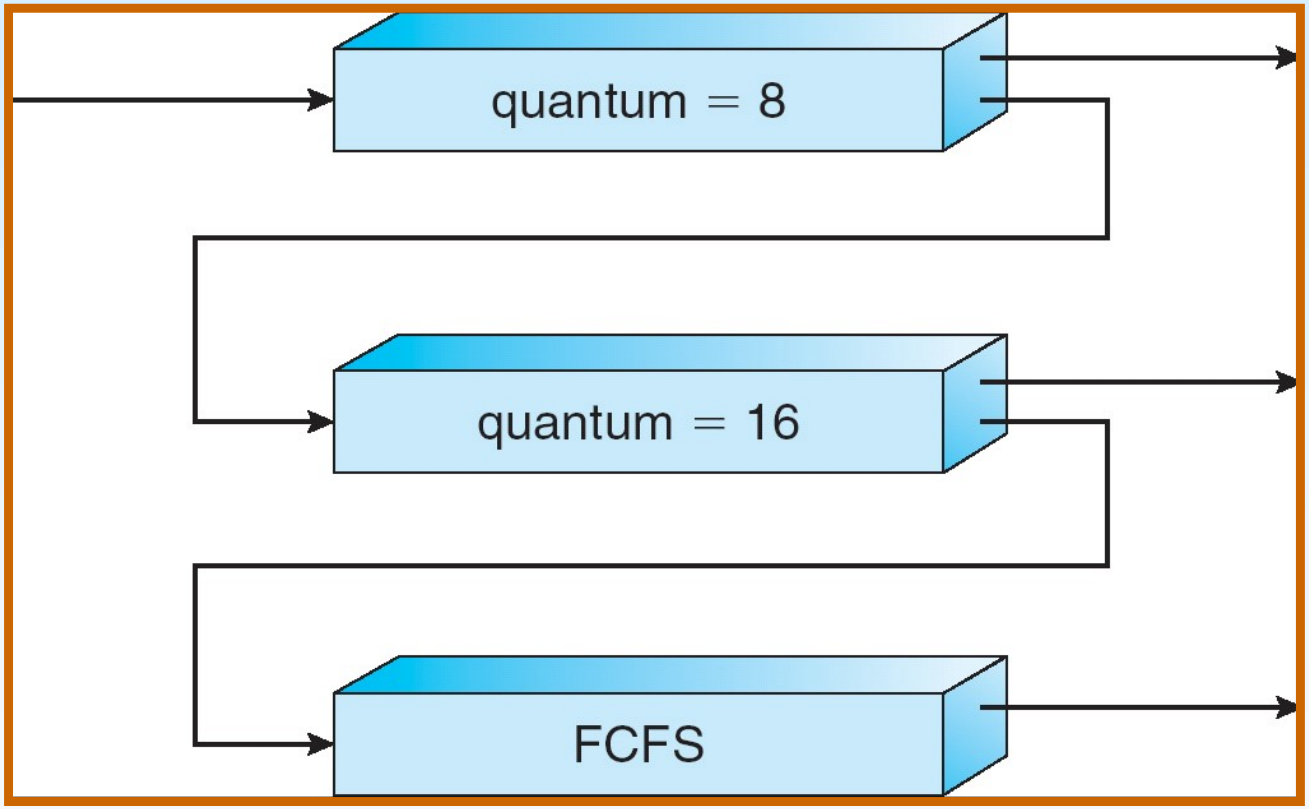
Multilevel Feedback Queue

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

Example of Multilevel Feedback Queue

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

Multilevel Feedback Queues



Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queueing models
- Implementation

Example of SRTF

- The arrival time and duration of the CPU and I/O bursts for each of the three processes A, B, and C are given in the table below. Each process has a CPU burst followed by an I/O burst followed by another CPU burst. Assume that each process has its own I/O resource.

Process	Arrival time	CPU burst	I/O burst	CPU burst
A	0	1	4	4
B	2	3	3	1
C	3	1	3	1

- The multi-programmed operating system uses the shortest remaining time first (SRTF) scheduling. What are the completion times of the processes A, B, and C and find individual waiting times of processes?

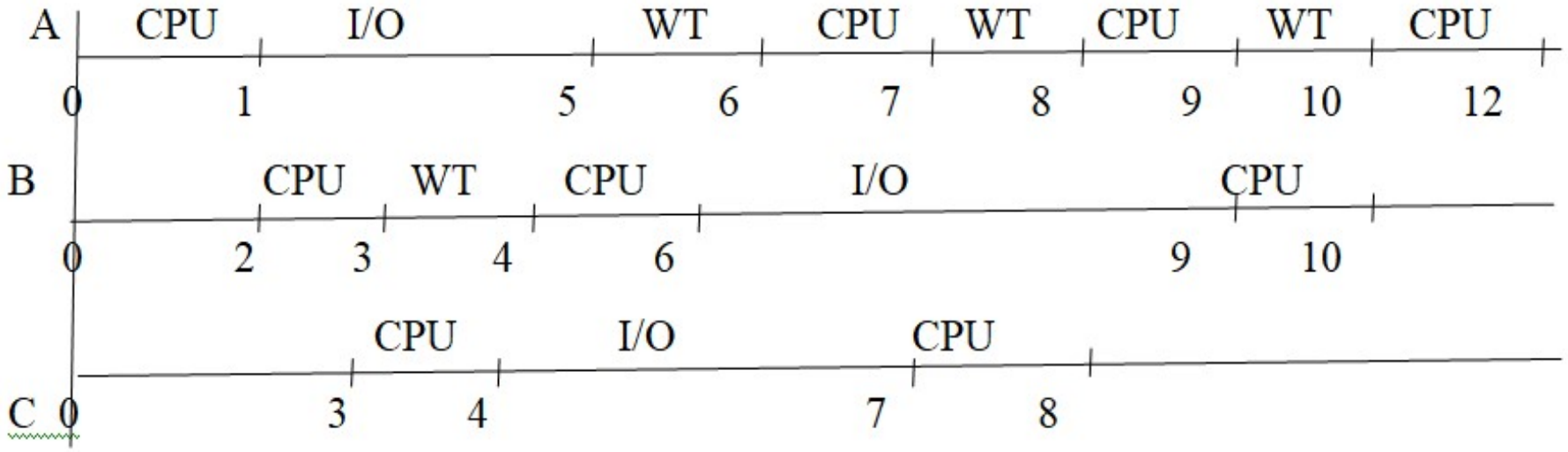
Example of SRTF

Process	Arrival Time	CPU Burst	I/O Burst	CPU Burst	Wait Time	Execution Time
A	0	1	4	4		
B	2	3	3	1		
C	3	1	3	1		


A
B
C

Process	Arrival time	CPU burst	I/O burst	CPU burst	Completion Time	Wait time
A	0	1, 0	4 (1 – 5)	4, 3, 2, 0	12	3
B	2	3, 2, 0	3 (6 – 9)	1, 0	10	1
C	3	1, 0	3 (4 – 7)	1, 0	8	0

Wait time (WT) :-



Process name	Waiting Time		Waiting Time (Total)
	Start Time	End Time	
A	5	6	3
	7	8	
	9	10	
B	3	4	1
C	Nil	Nil	0

Process name	Waiting Time		Waiting Time (Total)
	Start Time	End Time	
A	5	6	3
	7	8	
	9	10	
B	3	4	1
C	Nil	Nil	0

Consider three processes (P1, and P2, and P3), all arriving at time zero, with 10, 20, and 30 time units of execution respectively. P1 spends the first 50% of execution time doing I/O, the next 30% of time doing computation, and the last 20% of time doing I/O again, but P2 and P3 spend the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses a shortest remaining compute time first scheduling and schedules a new process either when the running process finishes its compute burst or when the running process gets blocked on I/O. Assume that all I/O operations can be overlapped. For what percentage of time does the CPU remain idle?

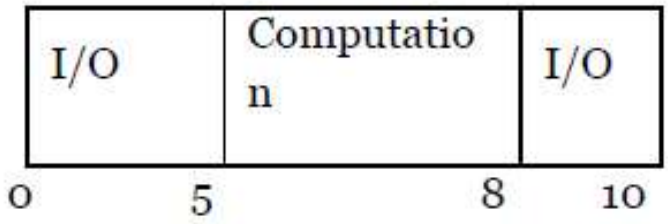
Process	Arrival Time	CPU Burst	Wait Time	Execution Time
P1	0	10		
P2	0	20		
P3	0	30		

P1 spends the
first 50% of execution time doing I/O,
the next 30% of time doing computation,
and the last 20% of time doing I/O again

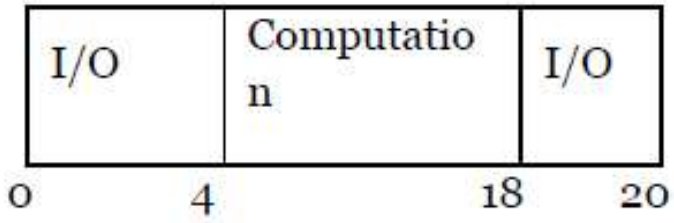
P2 and P3 spend the
first 20% of execution time doing I/O,
the next 70% of time doing computation,
and the last 10% of time doing I/O again.

0000-0003-1180-6782

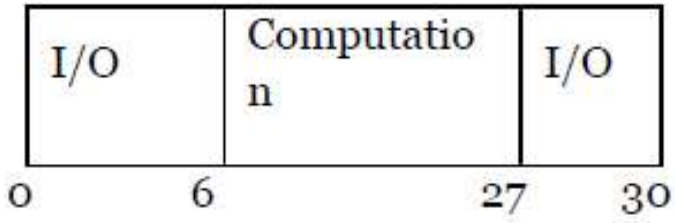
P1



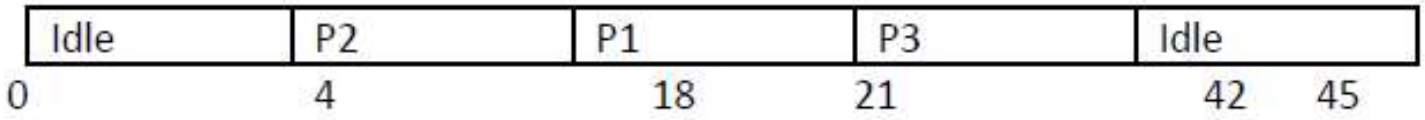
P2



P3



Execution of processes



Percentage of time CPU is idle=((4+3)/45)*100=15.5%

Use preemptive priority scheduling with aging technique to calculate average waiting time, average response time and average turnaround time of the processes as given below. The range of priority starts from 1 to 10, where 1 is the high priority and 10 is the low priority. Here, the criteria for aging technique is that the priority of the process will be increased by 1 if the process continuously waits in the ready queue for every 2 unit of the time. (For example, let process enters into the ready queue at time 11 with priority 6 and waits in ready queue till 13. As it has stayed continuously in ready queue for 2 units so at time 13, its priority changes from 6 to 5. If it waits in the ready queue continuously till 15, then its priority changes from 5 to 4 and so on.

Process	Arrival time	CPU burst	Priority
P1	7	2	5
P2	0	6	7
P3	4	3	5
P4	5	5	4
P5	8	4	3

Thread

Thread

- Why do we need a thread?
  Creating a new process is often very expensive.
  Processes don't (directly) share memory. Because each process has its own address space.

- What is a thread? (Figure 13)
  A thread is a light weight process. A thread defines a single sequential execution stream within a process (PC, stack, registers).
  Each thread has its own stack, PC, CPU registers, etc.
  All threads within a process share the same address space and OS resources.
  Threads share memory. They can communicate directly.

  Each thread has a thread control block (TCB). TCB contains processor state(Registers), thread state, and pointer to corresponding PCB.
  Context switch between two threads in the same process does not need to change address space.
  Context switch between two threads in different processes must change address space.

- Thread system classification (Figure 14)

- Different types of thread and their functionality.
  Basically there are two types :

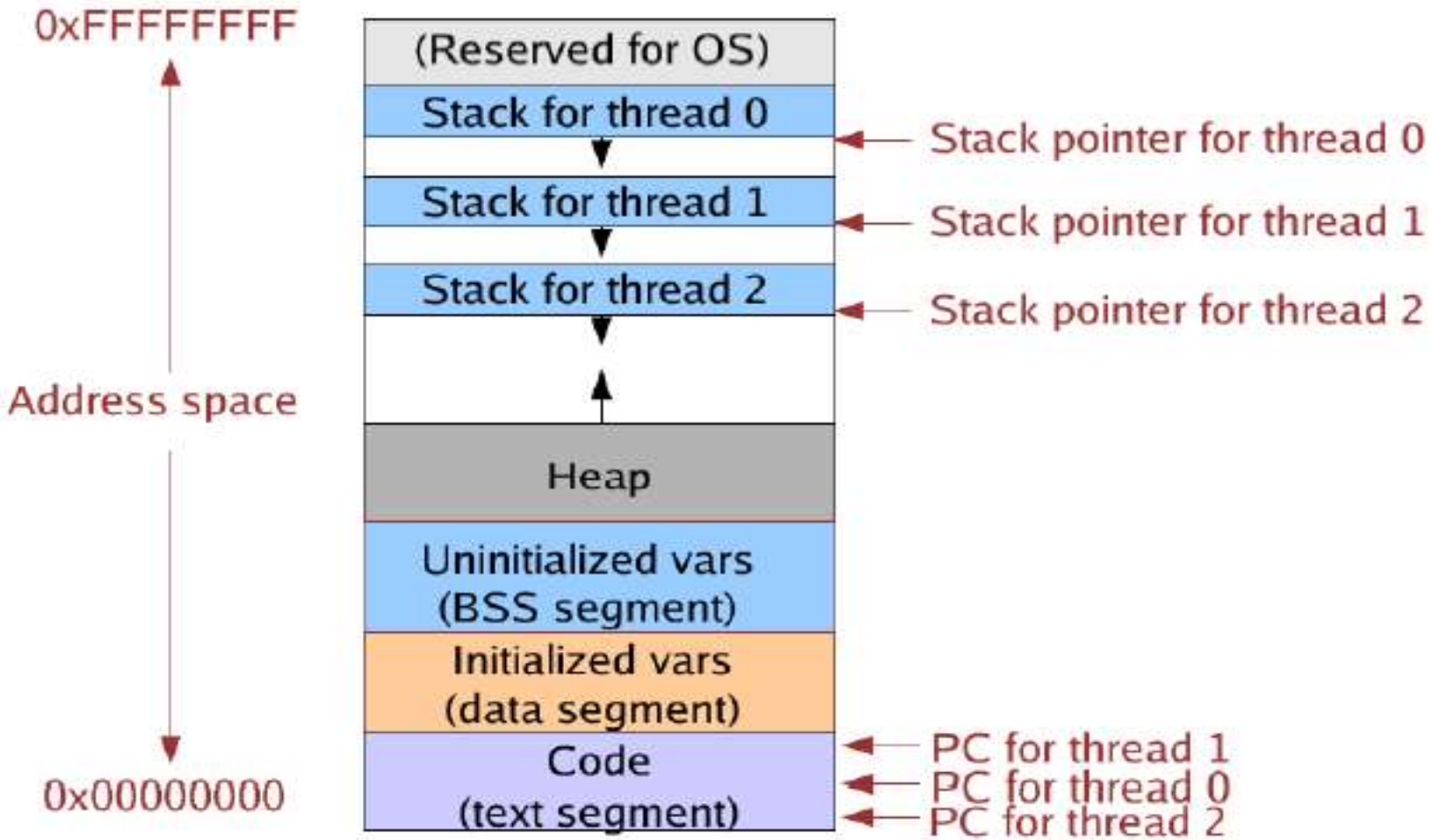


Figure 13. Thread

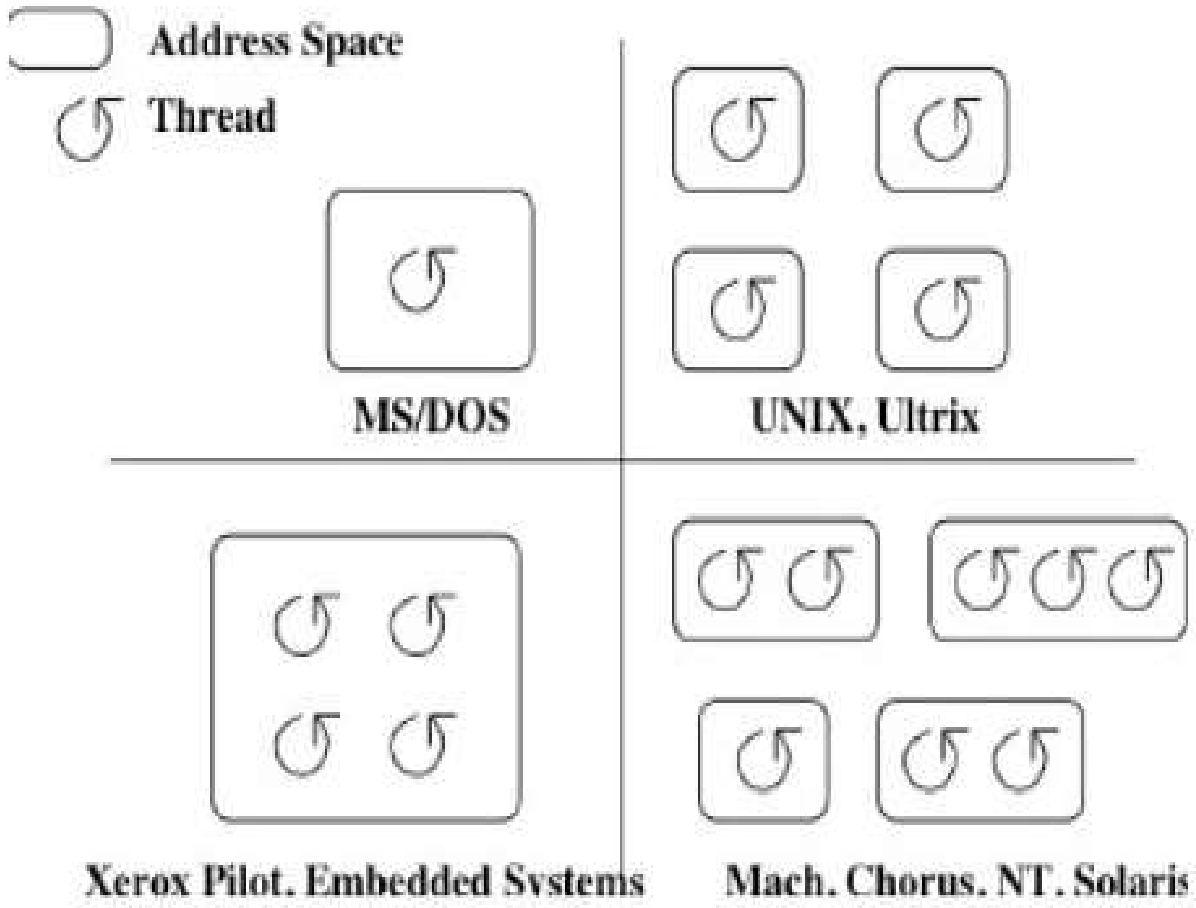


Figure 14. Thread system classification

- user level thread
- kernel level thread.

- User level thread (Figure 15)
  Some OS only supports user level thread. In this case, OS does not need to know anything about multiple threads in a process. So, managing multiple threads only requires switching the CPU state (PC, registers, etc.). And this can be done directly by a user program without the help of OS. The programmer uses a **thread library** to manage threads.

  - No context switching is needed.
  - Each process has one or more threads.
  - Each process may maintains a ready queue.
  - User level threads are more faster than other level threads.
  - OS finds all the treads belongs to one process is a single one.

  - The main disadvantages of user level thread are:

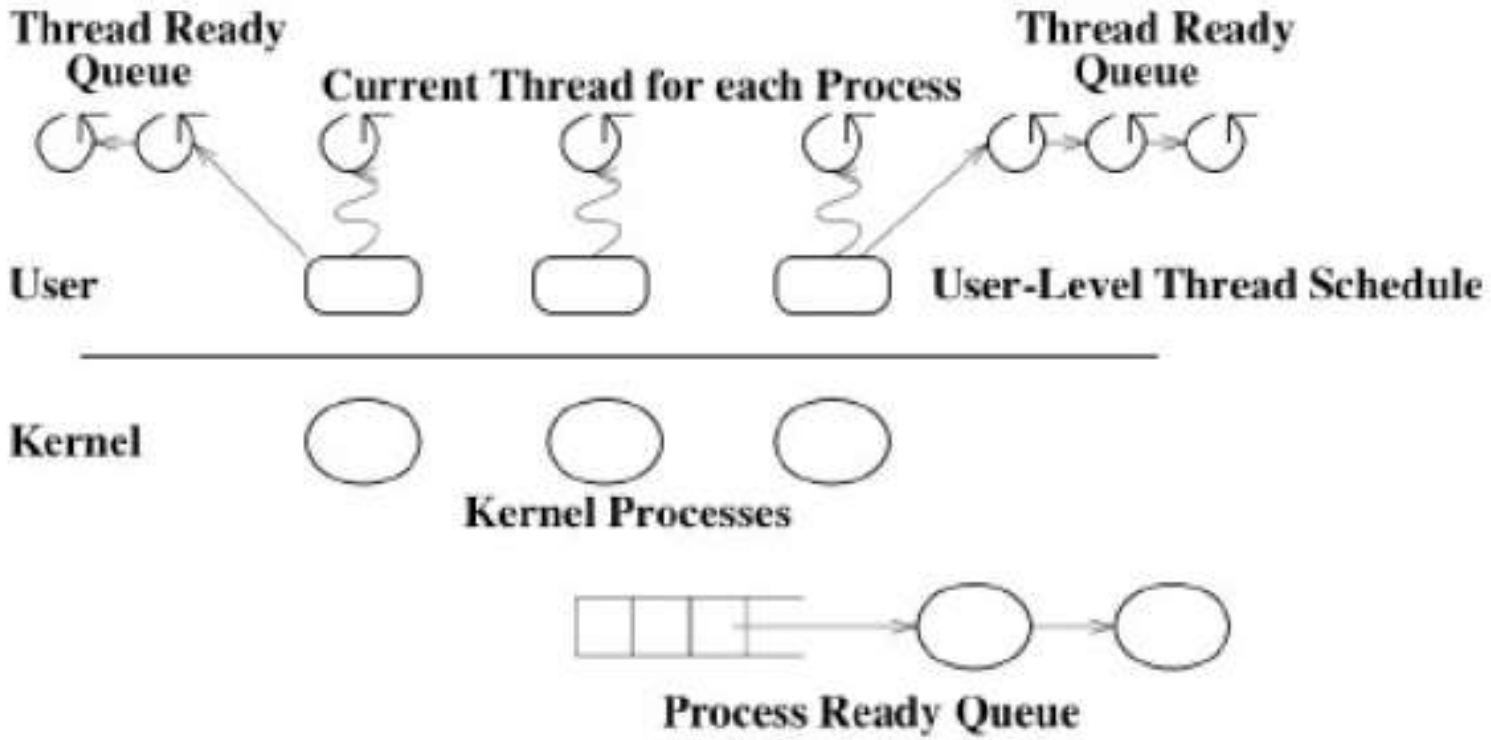


Figure 15. User level thread

    * if one thread wants for any I/O, then all the threads belongs to same process will automatically blocked.
    * Since the OS does not know about the existence of the user-level threads, it may make poor scheduling decisions.
  - How to create a user-level thread? (Figure 16)
    Thread library maintains a TCB for each thread in the application, just a linked list. Allocate a separate stack for each thread (usually with malloc).

- Kernel level thread
  - System call is used to create kernel thread.
  - A kernel thread is also known as a lightweight process. The kernel does thread creation, termination, joining, and scheduling in kernel space.
  - Kernel threads are usually slower than user threads due to system overhead.
  - Switching between kernel threads of the same process requires a small context switch.
    * The values of registers, program counter, and stack pointer must be changed.
    * Memory management information does not need to be changed since the threads share an address space.
  - The kernel must manage and schedule threads (as well as processes), but it can use the same process scheduling algorithms.
  - Switching between kernel threads is slightly faster than switching between processes.
  - In a multiprocessor environment, the kernel may schedule threads on different processors.

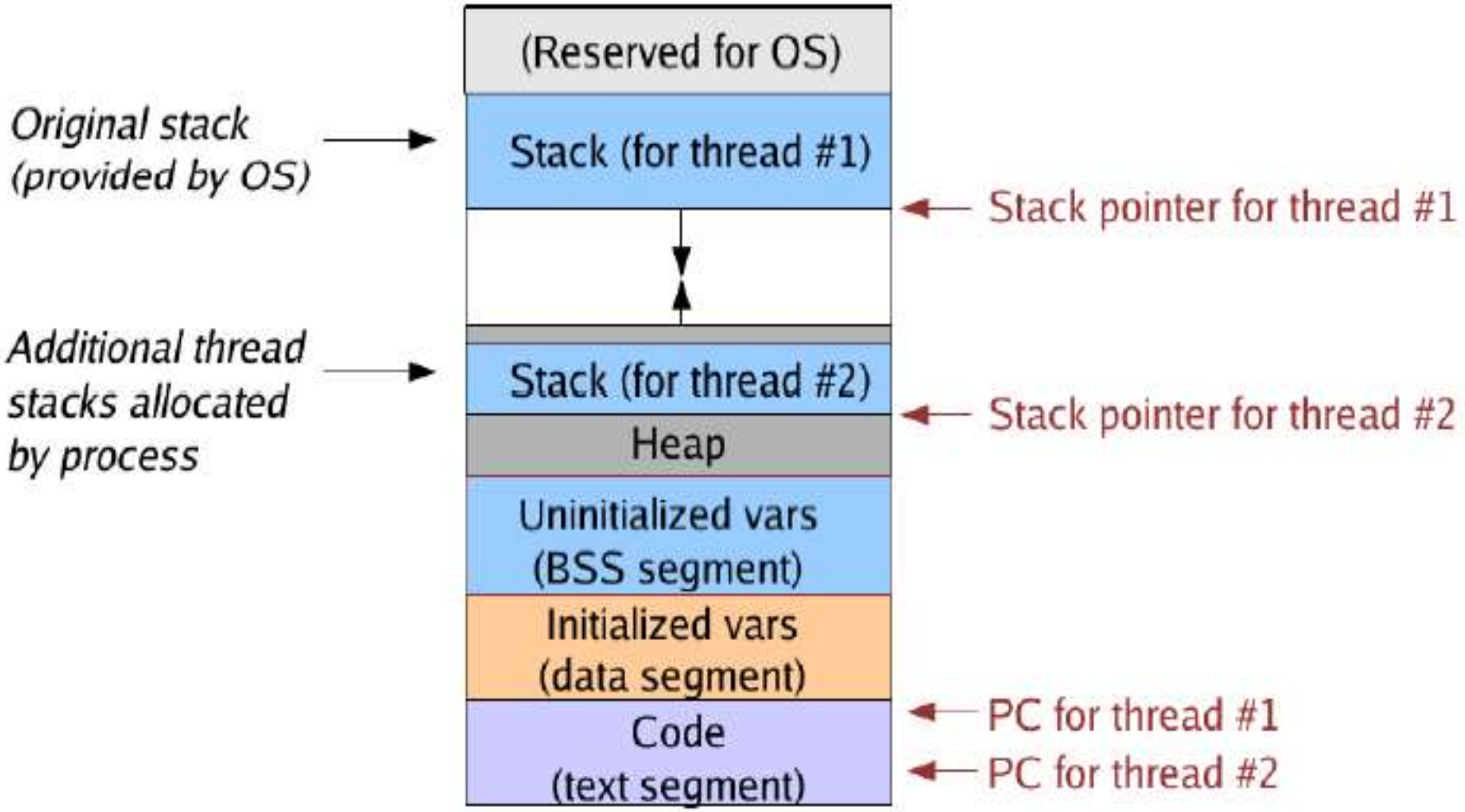
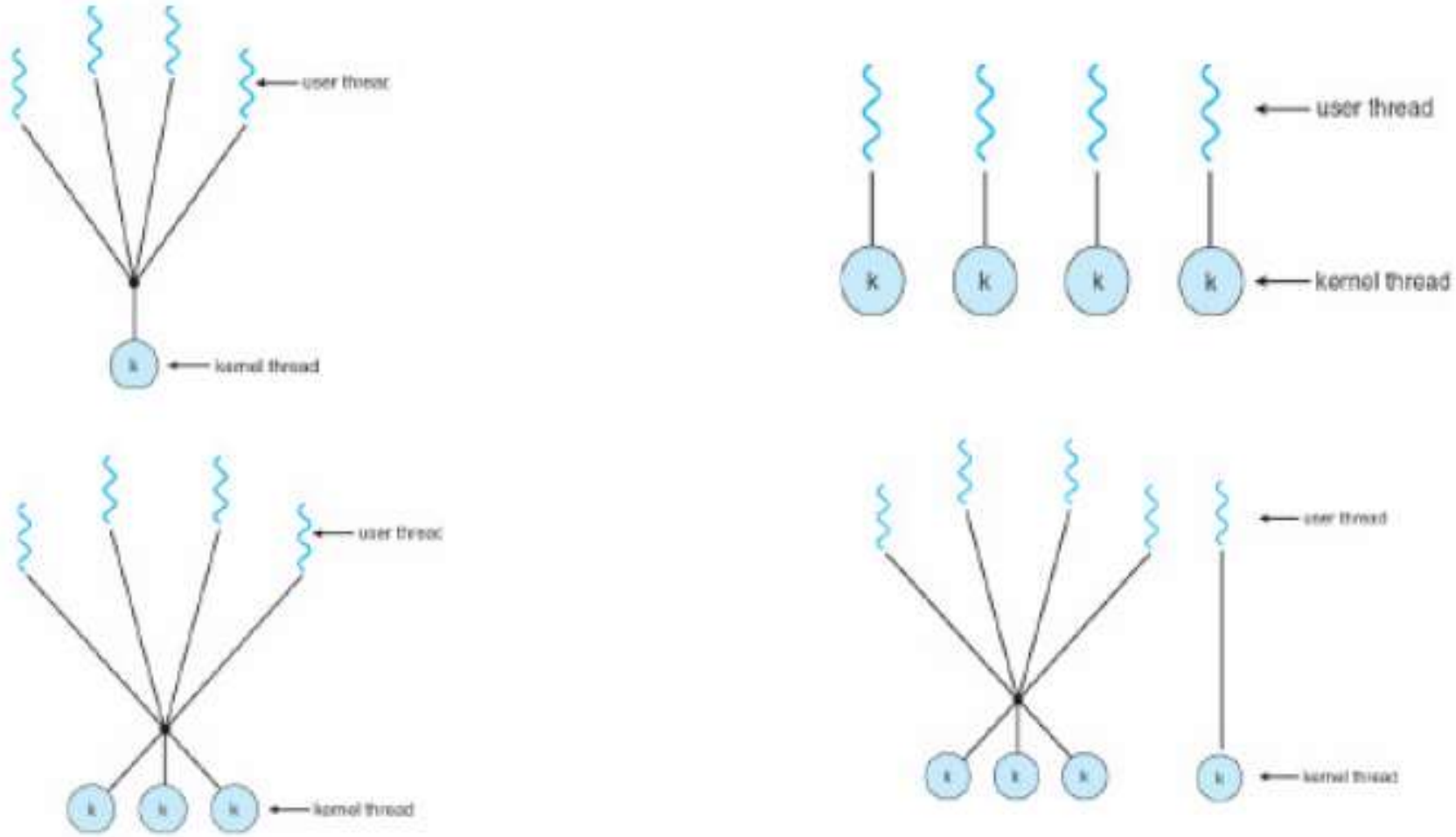


Figure 16. User Level Thread



Many-to-one, one-to-one, many-to-many and two-level

Figure 17. Thread model