OPERATING SYSTEM (ENCT254)

PROCESS MANAGEMENTT

Chapter-2 Getting Started..

Er. Sagar Panta
Asst. Lecturer, Dept. of Elx. & Computer Engineering
National College of Engineering
sagar@nce.edu.np

Process

- In general, a process is a program in execution.

- A Program is not a Process by default. A program is a passive entity, i.e. a file containing a list of instructions stored on disk (secondary memory) (often called an executable file).

- A program becomes a Process when an executable file is loaded into main memory and when it’s PCB is created

- A process on the other hand is an Active Entity, which require resources like main memory, CPU time, registers, system buses etc.

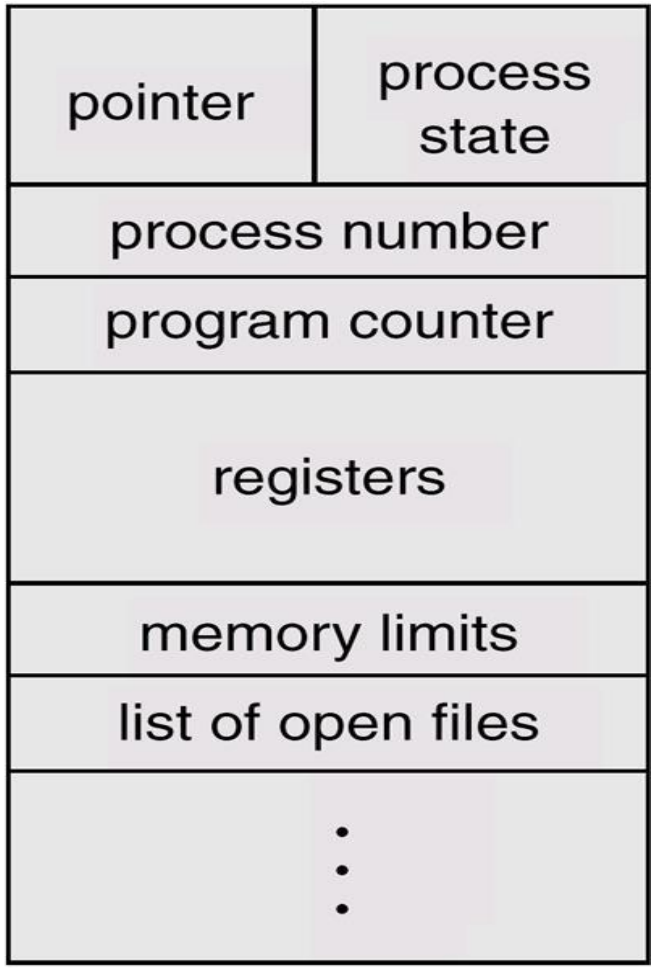
Process[Cont’d]

- Even if two processes may be associated with same program, they will be considered as two separate execution sequences and are totally different process.
- For instance, if a user has invoked many copies of web browser program, each copy will be treated as separate process. Even though the text section is same but the data, heap and stack sections can vary.

Process Control Block

- Each process is represented in the operating system by a process control block (PCB) — also called a task control block. PCB simply serves as the repository for any information that may vary from process to process.

- It contains many pieces of information associated with a specific process, including these:



Process Control Block[Cont’d]

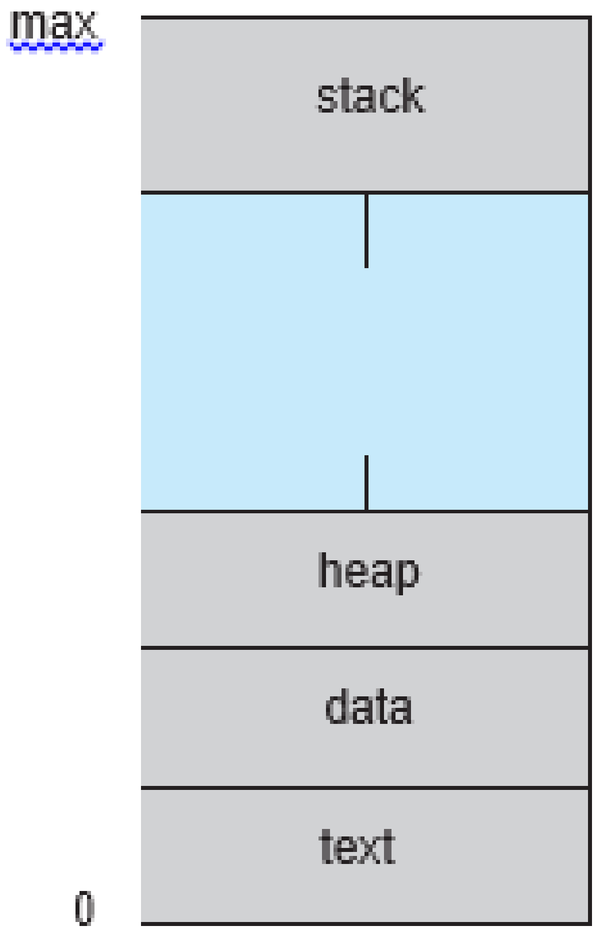
- Process state: The state may be new, ready, running, waiting, halted, and so on.
- Program counter: The counter indicates the address of the next instruction to be executed for this process.
- CPU registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

Process Control Block[Cont'd]

- CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- Memory-management information: This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
- Accounting information: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- I/O status information: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

A Process consists of following sections:

- **Text section:** also known as Program Code.
- **Stack:** which contains the temporary data (Function Parameters, return addresses and local variables).
- **Data Section:** containing global variables.
- **Heap:** which is memory dynamically allocated during process runtime.



Program Vs. Process

Program	Process
Consists of set of instructions in programming language	It is a sequence of instruction execution
It is a static object existing in a file form	It is a dynamic object (i.e. program in execution)
Program is loaded into secondary storage device	Process is loaded into main memory
The time span is unlimited	Time span is limited
It is a passive entity	It is an active entity

Operations on Process

- The user can perform the following operations on a process in the operating system:

  - Process Creation
  - Process scheduling or dispatching
  - Blocking
  - Preemption
  - Termination

Process Creation

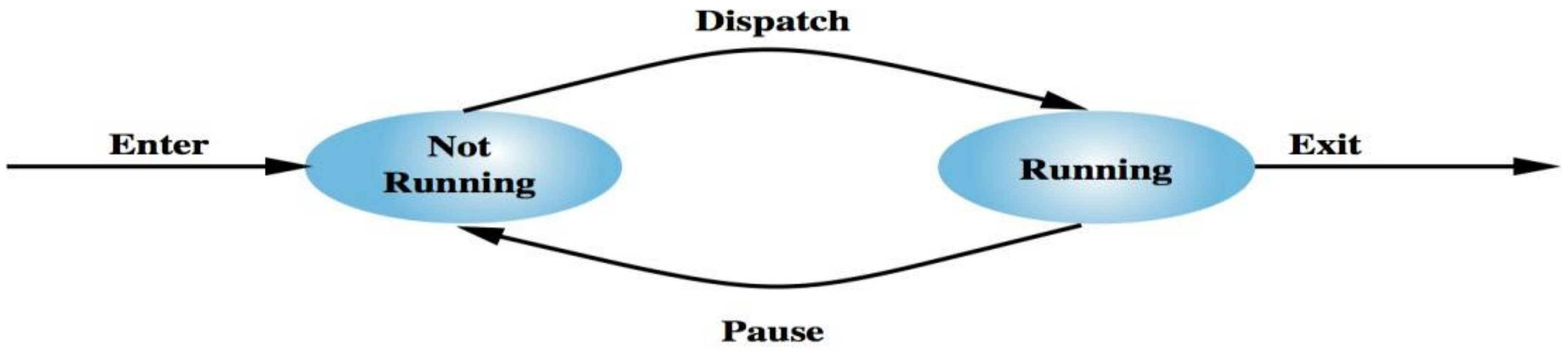
- It’s a job of OS to create a process. There are four ways achieving it:
  - For a batch environment a process is created in response to submission of a job.
  - In Interactive environment, a process is created when a new user attempt to log on.
  - The OS can create process to perform functions on the behalf a user program.
  - A number of process can be generated from the main process. For the purpose of modularity or to exploit parallelism a user can create numbers of process.

Process Termination

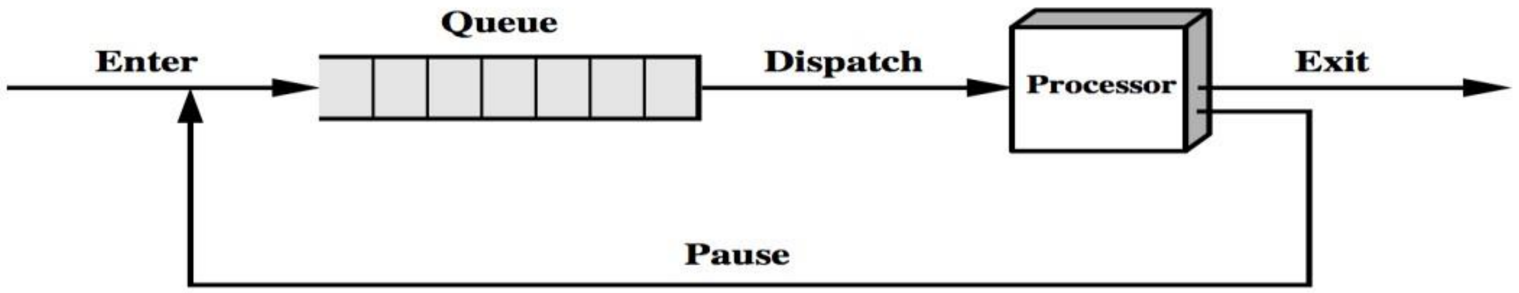
- A process terminates when it finishes executing its last statement. Its resources are returned to the system, it is purged from any system lists or tables, and its process control block (PCB) is erased. The new process terminates the existing process, usually due to following reasons:

  - **Normal Exit (Voluntary):** Most processes terminate because they have done their job.
  - **Error Exist(Voluntary):** When process discovers a fatal error. For example, a user tries to compile a program that does not exist.
  - **Fatal Error (Involuntary):** An error caused by process due to a bug in program for example, executing an illegal instruction, referring non-existing memory or dividing by zero.
  - **Killed by another Process (Industrial)**: A process executes a system call telling the Operating Systems to terminate some other process. In UNIX, this call is kill. In some systems when a process kills all processes it created are killed as well (UNIX does not work this way).

Proces State(2 State Model)



(a) State transition diagram



(b) Queuing diagram

- In this model, we consider two main states of the process. These two states are
- **State 1**: Process is Running on CPU
- **State 2**: Process is Not Running on CPU

- New: First of all, when a new process is created, then it is in Not Running State. Suppose a new process P2 is created then P2 is in NOT Running State.

- CPU: When CPU becomes free, Dispatcher gives control of the CPU to P2 that is in NOT Running state and waiting in a queue.

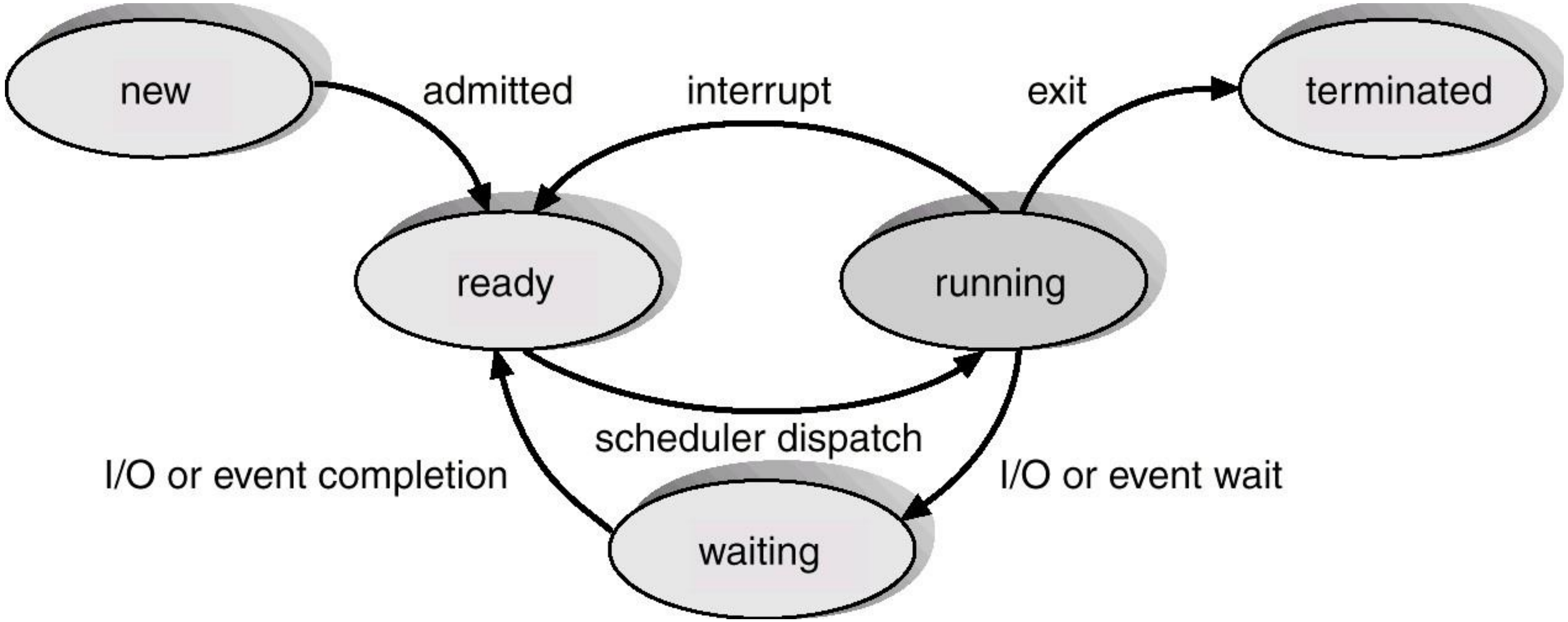
- Dispatcher: Dispatcher is a program that gives control of the CPU to the process selected by the CPU scheduler. Suppose dispatcher allow P2 to execute on CPU.

- Running: When dispatcher allows P2 to execute on CPU then P2 starts its execution. Here we can say that P2 is in running state.

- Now, if any process with high priority wants to execute on CPU, Suppose P3 with high priority, then P2 should be the pause or we can say that P2 will be in waiting state and P3 will be in running state.

- Now, when P3 terminates then P2 again allows the dispatcher to execute on CPU

Process State(5 State Model)



1. **New:** A process that has just been created but has not yet been admitted to the pool of executable processes by the OS.

2. **Ready:** Process that is prepared to execute when given the opportunity. That is, they are not waiting on anything except the CPU availability.

3. **Running:** the process that is currently being executed.

4. **Blocked:** A process that cannot execute until some event occurs, such as the completion of an I/O operation.

5. **Exit:** A process that has been released from the pool of executable processes by the OS, either because it is halted or because it is aborted for some reason A process that has been released by OS either after normal termination or after abnormal termination (error).

Scheduling

- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs.
- For achieving this, the scheduler must apply appropriate rules for swapping processes IN and out of CPU.

Scheduler

- Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run.

- **Types of Scheduler**
  - Long term scheduler
  - Mid - term scheduler
  - Short term scheduler

Assignment

- Long Term Scheduler
- Mid-term scheduler
- Short Term Scheduler

Dispatcher

- The dispatcher is the module that gives control of the CPU to the process selected by the short-time scheduler (selects from among the processes that are ready to execute).

- The function involves:
  - ➢Context Switching
  - ➢Switching to user mode
  - ➢Restart the execution of process

Context Switching

- A context switch is the mechanism to store and restore the state of a Process in Process Control block so that a process execution can be resumed from the same point at a later time.
- Using this technique, a context switcher enables multiple processes to share a single CPU.
- Context switching is an essential part of a multitasking operating system features.
- When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block.
- After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

Process scheduling Types

- In an operating system (OS), a process scheduler performs the important activity of scheduling a process between the ready queues and waiting queue and allocating them to the CPU.

- The OS assigns priority to each process and maintains these queues.

- The scheduler selects the process from the queue and loads it into memory for execution.

- There are two types of process scheduling:
    - Preemptive scheduling
    - Non-preemptive scheduling.

1. Preemptive Scheduling

- The scheduling in which a running process can be interrupted if a high priority process enters the queue and is allocated to the CPU is called preemptive scheduling.

- In this case, the current process switches from the running queue to ready queue and the high priority process utilizes the CPU cycle.

2. Non-Preemptive Scheduling

- The scheduling in which a running process cannot be interrupted by any other process is called non-preemptive scheduling.

- Any other process which enters the queue has to wait until the current process finishes its CPU cycle.

Preemptive Vs. Non- Preemptive Scheduling

PREEMPTIVE SCHEDULING	NON-PREEMPTIVE SCHEDULING
The resources are allocated to a process for a limited time.	Once resources are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Process can be interrupted in between.	Process cannot be interrupted till it terminates or switches to waiting state.
If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Preemptive scheduling has overheads of scheduling the processes.	Non-preemptive scheduling does not have overheads.
Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.
Preemptive scheduling is cost associated.	Non-preemptive scheduling is not cost associative.

Scheduling Algorithms

- The various scheduling algorithms are as follows:

First Come First Served (FCFS) Scheduling

- It is simplest CPU scheduling algorithm.

- The FCFS scheduling algorithm is non preemptive i.e. once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

- In this technique, the process that requests the CPU first is allocated the CPU first.

- When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

- The average waiting time under this technique is often quite long.

Shortest Job First (SJF) Scheduling

- This technique is associated with the length of the next CPU burst of a process.

- When the CPU is available, it is assigned to the process that has smallest next CPU burst. If the next bursts of two processes are the same, FCFS scheduling is used.

- The SJF algorithm is optimal i.e. it gives the minimum average waiting time for a given set of processes.

- The real difficulty with this algorithm knows the length of next CPU request.

Shortest Remaining Time (SRT) Scheduling

- A preemptive SJF algorithm will preempt the currently executing, where as a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is also known **Shortest-Remaining-time (SRT) First Scheduling.**

- It is a preemptive version of SJF algorithm where the remaining processing time is considered for assigning CPU to the next process.

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

Round Robin Scheduling

- Round Robin (RR) scheduling algorithm is designed especially for time sharing systems. It is similar to FCFS scheduling but preemption is added to enable the system to switch between processes. A small unit of time called **time quantum** or **time slice** is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum.
- The process may have a CPU burst less than 1-time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1-time quantum, the timer will go off and will cause an interrupt to the OS. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.
- The average waiting time under the RR policy is often long.

Highest-Response Ratio Next (HRN) Scheduling

- Highest Response Ratio Next (HRRN) scheduling is a non-preemptive discipline, in which the priority of each job is dependent on its estimated run time, and also the amount of time it has spent waiting. Jobs gain higher priority the longer they wait, which prevents indefinite postponement (process starvation).

- It selects a process with the largest ratio of waiting time over service time. This guarantees that a process does not starve due to its requirements.

- In fact, the jobs that have spent a long time waiting compete against those estimated to have short run times.

Response Ratio = (waiting time + service time) / service time

Advantages

- Improves upon SJF scheduling
- Still non-preemptive
- Considers how long process has been waiting
- Prevents indefinite postponement

Terminologies

- **Arrival Time (AT):** Time at which process enters a ready state.
- **Burst Time (BT):** Amount of CPU time required by the process to finish its execution.
- **Completion Time (CT):** Time at which process finishes its execution.
- **Turn Around Time (TAT):** Completion Time (CT) – Arrival Time (AT), WT + BT
- **Waiting Time(WT):** Turn Around Time (TAT) – Burst Time (BT)
- **Average turnaround time: (∑TAT)**/No. of Processes
- **Average Waiting Time: (∑WT)**/No. of Processes

NUMERICALS

1. Schedule the following set of process according to:
   i. FCFS
   ii. SJF
   iii. SRTN
   iv. RR(time quantum = 4 ms)
   v. HRRN

   And calculate **average waiting time** and **average turnaround time**.

Process	Arrival Time(AT)	CPU Time(CT)
A	0	12
B	2	8
C	5	7
D	10	9

Process	Arrival Time(AT)	CPU Time(CT)
A	0	12
B	2	8
C	5	7
D	10	9

According to FCFS

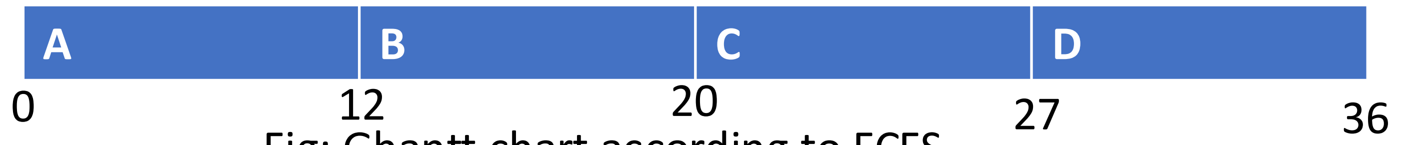


Fig: Ghantt chart according to FCFS

Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
A	0	12	12	12	0
B	2	8	20	18	10
C	5	7	27	22	15
D	10	9	36	26	17
∑				78	42

Average TAT = ∑TAT/no. of process = 78/4 = 19.5 ms

Average WT = ∑WT/no. of process = 42/4 = 10.5 ms

Process	Arrival Time(AT)	CPU Time(CT)
A	0	12
B	2	8
C	5	7
D	10	9

According to SJF

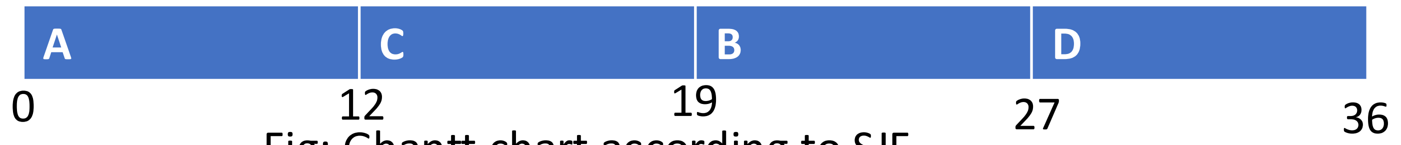


Fig: Ghantt chart according to SJF

Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
A	0	12	12	12	0
B	2	8	27	25	17
C	5	7	19	14	7
D	10	9	36	26	17
∑				77	41

Average TAT = ∑TAT/no. of process = 77/4 = 19.25 ms
Average WT = ∑WT/no. of process = 41/4 = 10.25 ms

Proces s	Arrival Time(AT)	CPU Time(CT)
A	0	12
B	2	8
C	5	7
D	10	9

According to SRTN

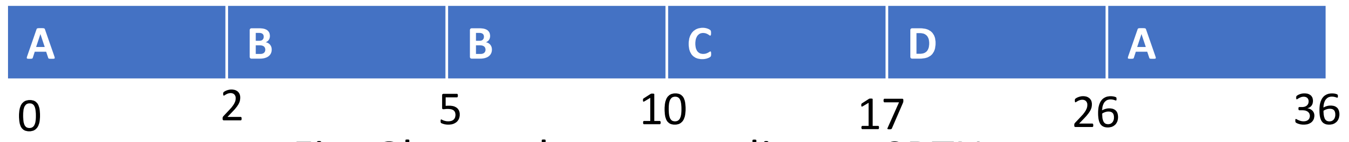


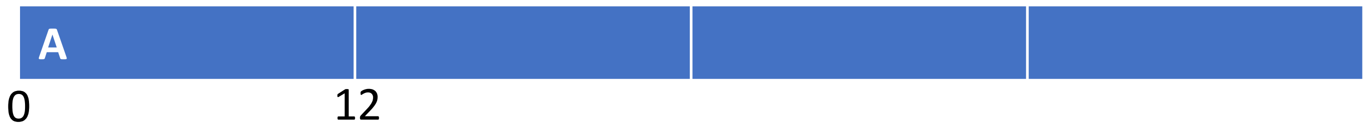
Fig: Ghantt chart according to SRTN

Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
A	0	12	36	36	24
B	2	8	10	8	0
C	5	7	17	12	5
D	10	9	26	16	7
∑				72	36

Average TAT = ∑TAT/no. of process = 72/4 = 18 ms
Average WT = ∑WT/no. of process = 36/4 = 9 ms

According to HRRN

Proces s	Arrival Time(AT)	CPU Time(CT)
A	0	12
B	2	8
C	5	7
D	10	9



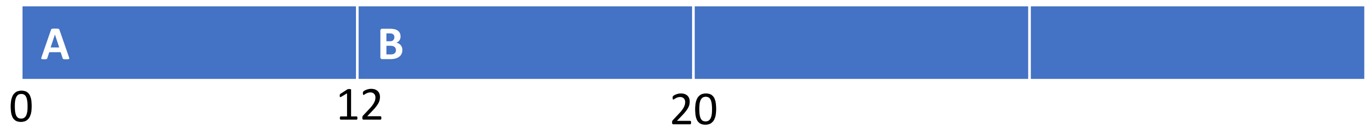
After 12ms,

(Response Ratio)B = (WT+ ST)/ST = (10+8)/8 =2.25
(Response Ratio)C = (WT+ ST)/ST = (7+7)/7 =2
(Response Ratio)D = (WT+ ST)/ST = (2+9)/9 = 1.22

Here, Process B has highest response ratio.
Hence, Process B will come after process A in Ghantt Chart

Proces s	Arrival Time(AT)	CPU Time(CT)
A	0	12
B	2	8
C	5	7
D	10	9

According to HRRN



After 20ms,

(Response Ratio)C = (WT+ ST)/ST = (15+7)/7 =3.14
(Response Ratio)D = (WT+ ST)/ST = (10+9)/9 = 1.88

Here, Process C has highest response ratio.
Hence, Process C will come after process B and remaining process D in Ghantt Chart.

According to HRRN

Process	Arrival Time(AT)	CPU Time(CT)
A	0	12
B	2	8
C	5	7
D	10	9

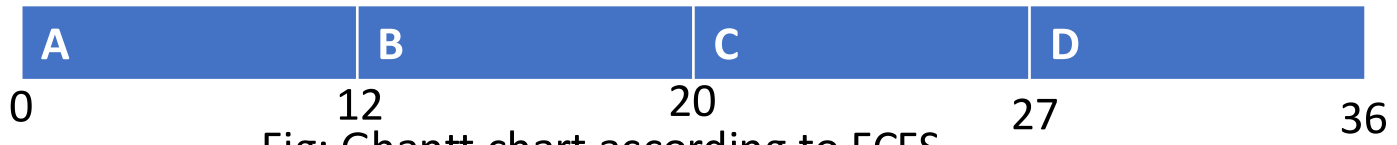


Fig: Ghantt chart according to FCFS

Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
A	0	12	12	12	0
B	2	8	20	18	10
C	5	7	27	22	15
D	10	9	36	26	17
∑				78	42

Average TAT = ∑TAT/no. of process = 78/4 = 19.5 ms
Average WT = ∑WT/no. of process = 42/4 = 10.5 ms

According to RR (Time quantum =4ms)

Process	Arrival Time(AT)	CPU Time(CT)
A	0	12
B	2	8
C	5	7
D	10	9

- Ready Queue

- Ghantt Chart

According to RR

Process	Arrival Time(AT)	CPU Time(CT)
A	0	12
B	2	8
C	5	7
D	10	9

- Ready Queue

- Ghantt Chart

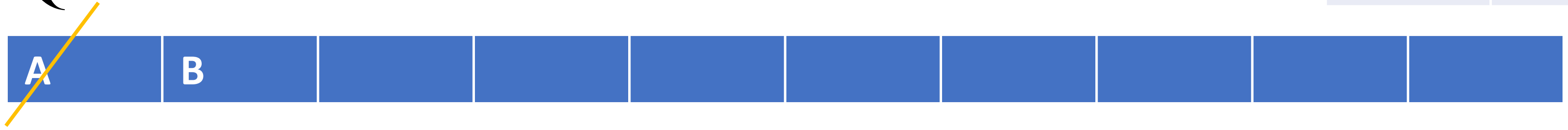
Fig: Ghantt chart according to RR

A arrives in ready queue at 0ms.

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ 8
B	2	8
C	5	7
D	10	9

According to RR

- Ready Queue



- Ghantt Chart

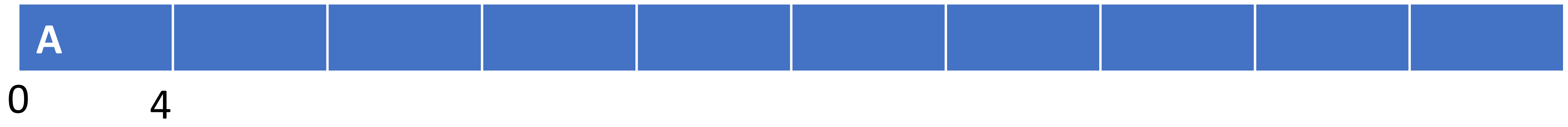


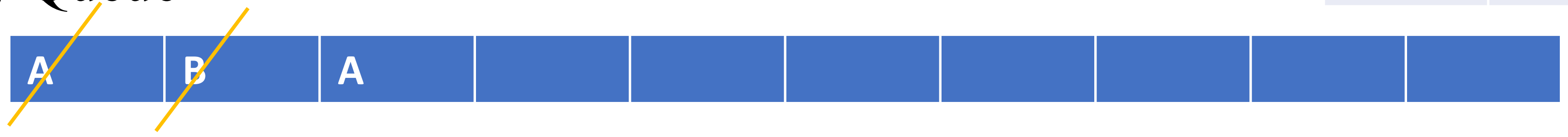
Fig: Ghantt chart according to RR

Since only A is arrived, A gets the CPU for 4 ms(quantum time).
At time A completes its 4ms quantum time, Process B arrives in ready queue.

According to RR

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ 8
B	2	~~8~~ 4
C	5	7
D	10	9

- Ready Queue



- Ghantt Chart

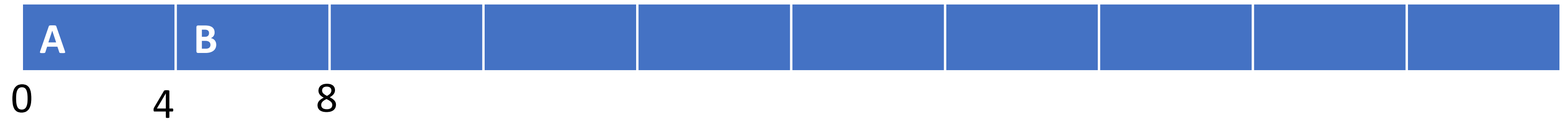


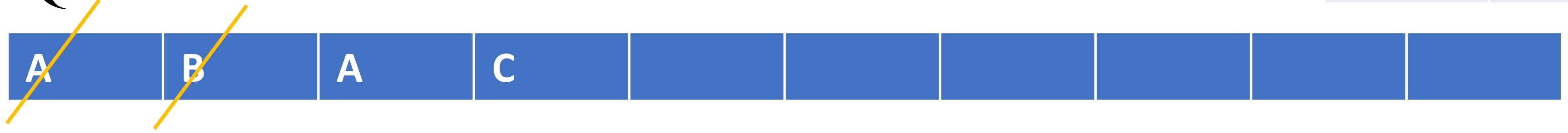
Fig: Ghantt chart according to RR

So, Process A is preempted and B gets CPU time for 4ms(quantum time).
As A has not finished executing(8ms remaining), A enters ready queue again.

Proces s	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ 8
B	2	~~8~~ 4
C	5	7
D	10	9

According to RR

- Ready Queue



- Ghantt Chart

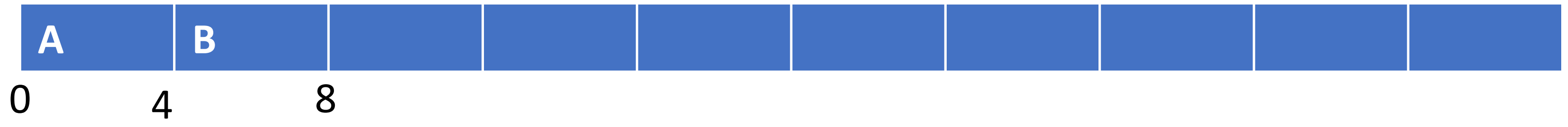


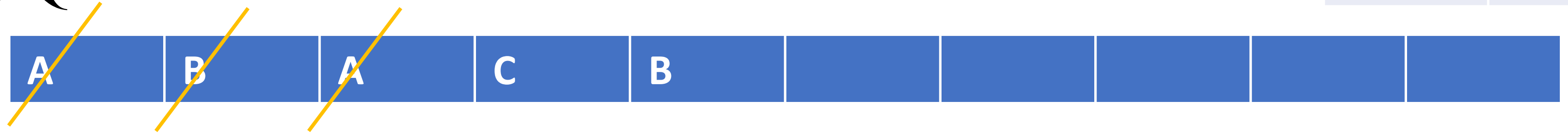
Fig: Ghantt chart according to RR

At time B completes its 4ms quantum time(i.e. after 8ms), Process C arrives in ready queue.

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ ~~8~~ 4
B	2	~~8~~ 4
C	5	7
D	10	9

According to RR

- Ready Queue



- Ghantt Chart

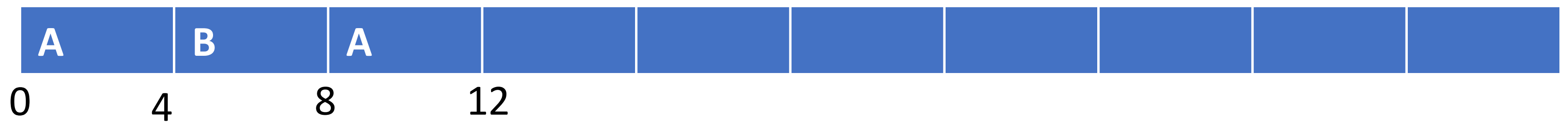


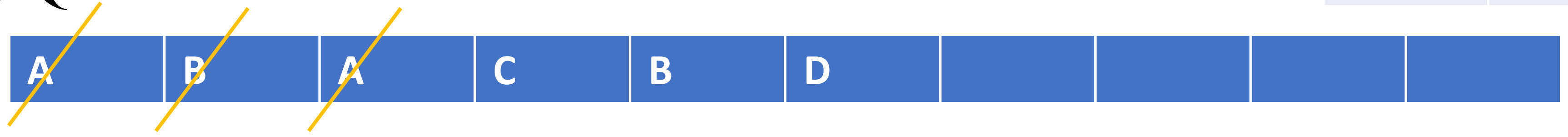
Fig: Ghantt chart according to RR

Process B is preempted and A gets CPU time for 4ms(quantum time).
As B has not finished executing(4ms remaining), B enters ready queue again.

According to RR

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ ~~8~~ 4
B	2	~~8~~ 4
C	5	7
D	10	9

- Ready Queue



- Ghantt Chart

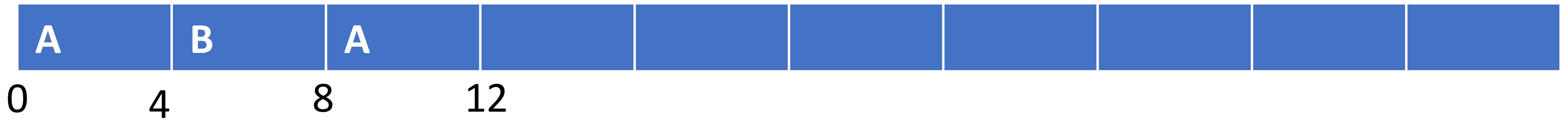


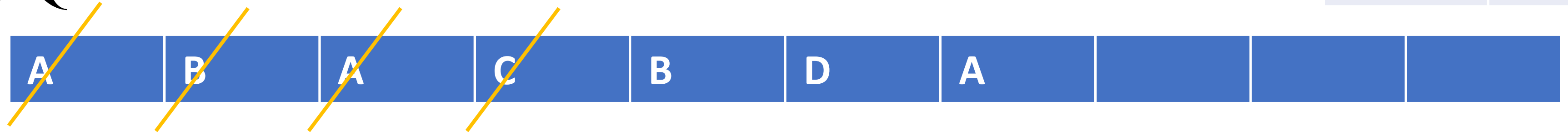
Fig: Ghantt chart according to RR

At time A completes its 4ms quantum time(i.e. after 12ms), Process D arrives in ready queue.

According to RR

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ ~~8~~ 4
B	2	~~8~~ 4
C	5	~~7~~ 3
D	10	9

- Ready Queue



- Ghantt Chart

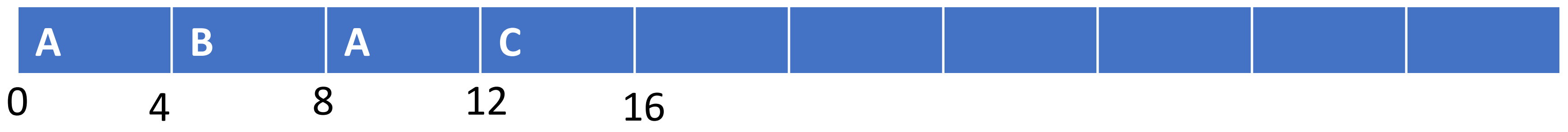


Fig: Ghantt chart according to RR

Process A is preempted and C gets CPU time for 4ms(quantum time).
As A has not finished executing(4ms remaining), A enters ready queue again.

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ ~~8~~ 4
B	2	~~8~~ ~~4~~
C	5	~~7~~ 3
D	10	9

According to RR

- Ready Queue

~~A~~	~~B~~	~~A~~	~~C~~	~~B~~	D	A	C		

- Ghantt Chart

A	B	A	C	B					

0 4 8 12 16

Fig: Ghantt chart according to RR

Process C is preempted and B gets CPU time for 4ms(quantum time).
As C has not finished executing(3ms remaining), C enters ready queue again.

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ ~~8~~ 4
B	2	~~8~~ ~~4~~
C	5	~~7~~ 3
D	10	~~9~~ 5

According to RR

- Ready Queue

~~A~~	~~B~~	~~A~~	~~C~~	~~B~~	~~D~~	A	C		

- Ghantt Chart

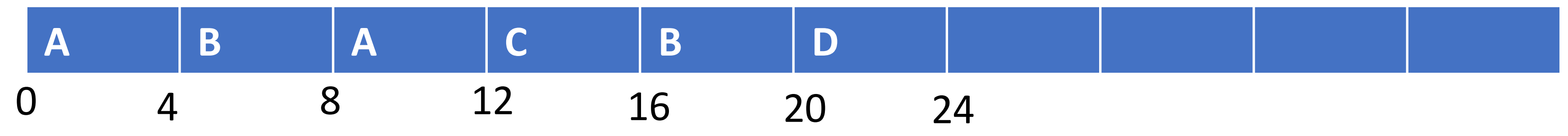


Fig: Ghantt chart according to RR

Process B is preempted and D gets CPU time for 4ms(quantum time).
Here B has completed its execution.

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ ~~8~~ ~~4~~
B	2	~~8~~ ~~4~~
C	5	~~7~~ 3
D	10	~~9~~ 5

According to RR

- Ready Queue

~~A~~	~~B~~	~~A~~	~~C~~	~~B~~	~~D~~	~~A~~	C	D	

- Ghantt Chart

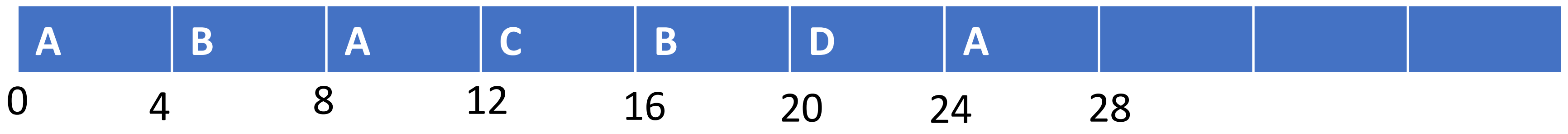


Fig: Ghantt chart according to RR

Process D is preempted and A gets CPU time for 4ms(quantum time).
Here A has completed its execution.

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ ~~8~~ ~~4~~
B	2	~~8~~ ~~4~~
C	5	~~7~~ ~~3~~
D	10	~~9~~ 5

According to RR

- Ready Queue

~~A~~	~~B~~	~~A~~	~~C~~	~~B~~	~~D~~	~~A~~	~~C~~	D	

- Ghantt Chart

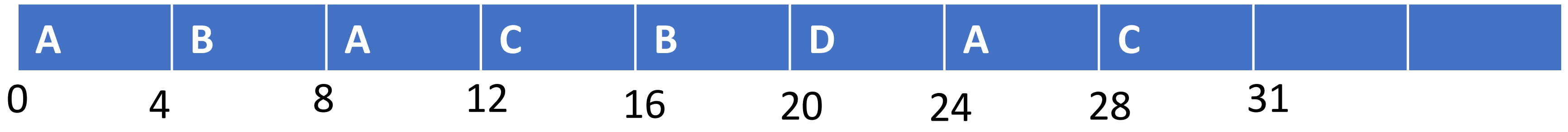


Fig: Ghantt chart according to RR

Process A has finished executing and C takes CPU time of 3ms to finish its execution. Here C completes its execution.

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ ~~8~~ ~~4~~
B	2	~~8~~ ~~4~~
C	5	~~7~~ ~~3~~
D	10	~~9~~ ~~5~~ 1

According to RR

- Ready Queue

~~A~~	~~B~~	~~A~~	~~C~~	~~B~~	~~D~~	~~A~~	~~C~~	~~D~~	

- Ghantt Chart

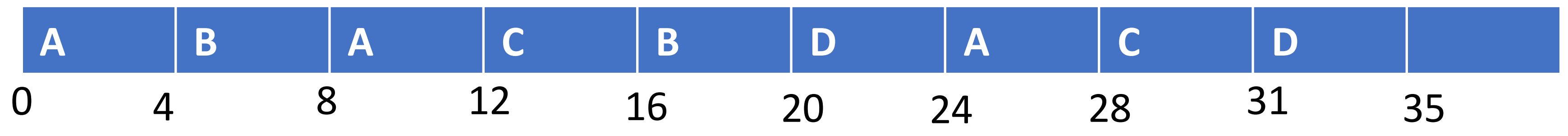


Fig: Ghantt chart according to RR

Process C has finished executing and D gets CPU time for 4ms(quantum time).

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ ~~8~~ ~~4~~
B	2	~~8~~ ~~4~~
C	5	~~7~~ ~~3~~
D	10	~~9~~ ~~5~~ 1

According to RR

- Ready Queue

~~A~~	~~B~~	~~A~~	~~C~~	~~B~~	~~D~~	~~A~~	~~C~~	~~D~~	D

- Ghantt Chart

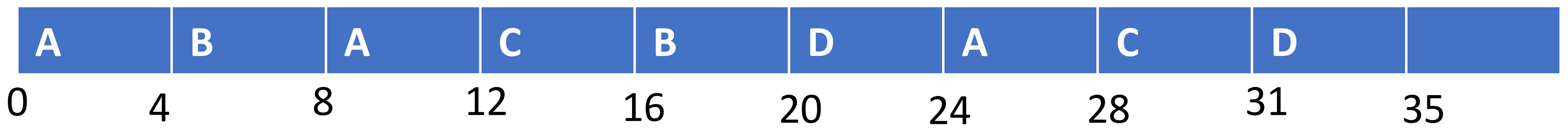


Fig: Ghantt chart according to RR

D hasnot still finished executing.
Hence D enters ready queue again after 4ms quantum.

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ ~~8~~ ~~4~~
B	2	~~8~~ ~~4~~
C	5	~~7~~ ~~3~~
D	10	~~9~~ ~~5~~ ~~1~~

According to RR

- Ready Queue

~~A~~	~~B~~	~~A~~	~~C~~	~~B~~	~~D~~	~~A~~	~~C~~	~~D~~	~~D~~

- Ghantt Chart

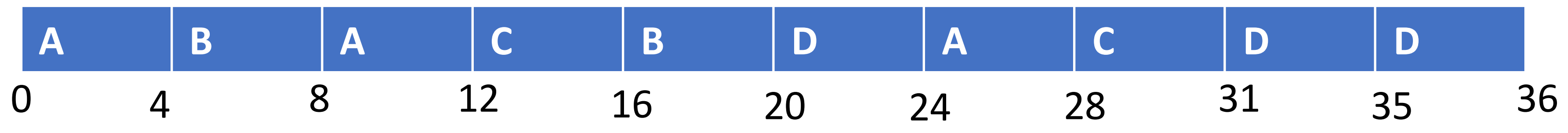


Fig: Ghantt chart according to RR

D gets CPU time.
D finishes executing.

According to RR

Process	Arrival Time(AT)	CPU Time(CT)
A	0	~~12~~ ~~8~~ ~~4~~
B	2	~~8~~ ~~4~~
C	5	~~7~~ ~~3~~
D	10	~~9~~ ~~5~~ ~~1~~

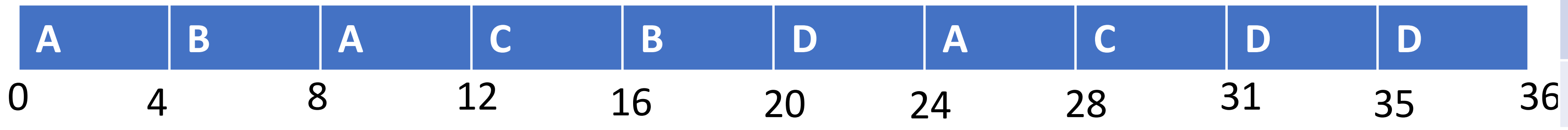


Fig: Ghantt chart according to RR

Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
A	0	12	28	28	16
B	2	8	20	18	10
C	5	7	31	26	19
D	10	9	36	26	17
∑				98	62

Average TAT = ∑TAT/no. of process = 98/4 = 24.5 ms
Average WT = ∑WT/no. of process = 62/4 = 15.5 ms

NUMERICALS

2. Schedule the following set of process according to:
   i. FCFS
   ii. SJF
   iii. SRTN
   iv. RR(time quantum = 2 ms)

   And calculate **average waiting time** and **average turnaround time**.

Process	Arrival Time(ms)	CPU Time(ms)
P1	0	3
P2	1	5
P3	2	7
P4	3	1

Proces s	Arrival Time(AT)	CPU Time(CT)
P1	0	3
P2	1	5
P3	2	7
P4	3	1

According to FCFS

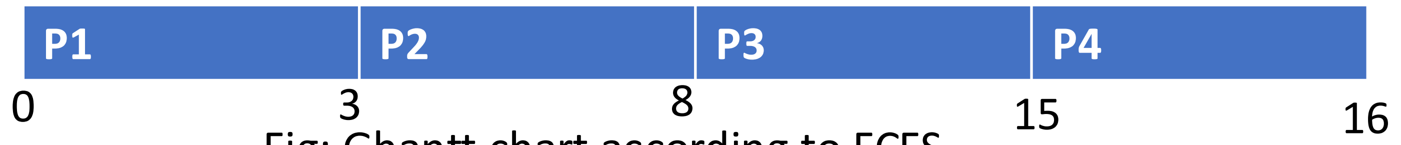


Fig: Ghantt chart according to FCFS

Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
P1	0	3	3	3	0
P2	1	5	8	7	2
P3	2	7	15	13	6
P4	3	1	16	13	12
∑				36	20

Average TAT = ∑TAT/no. of process = 36/4 = 9 ms
Average WT = ∑WT/no. of process = 20/4 = 5 ms

Proces s	Arrival Time(AT)	CPU Time(CT)
P1	0	3
P2	1	5
P3	2	7
P4	3	1

According to SJF

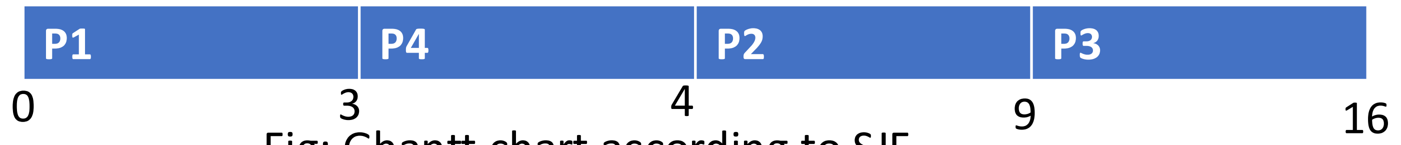


Fig: Ghantt chart according to SJF

Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
P1	0	3	3	3	0
P2	1	5	9	8	3
P3	2	7	16	14	7
P4	3	1	4	1	0
∑				26	10

Average TAT = ∑TAT/no. of process = 26/4 = 6.5 ms
Average WT = ∑WT/no. of process = 10/4 = 2.5 ms

Process	Arrival Time(AT)	CPU Time(CT)
P1	0	3
P2	1	5
P3	2	7
P4	3	1

According to SRTN

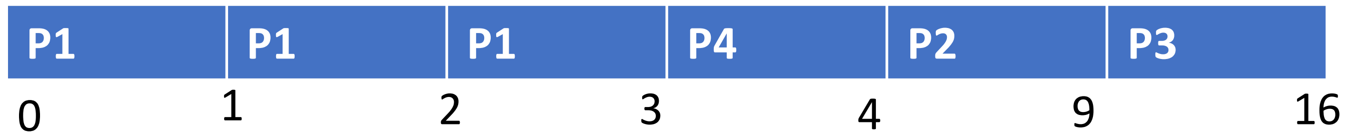


Fig: Ghantt chart according to SRTN

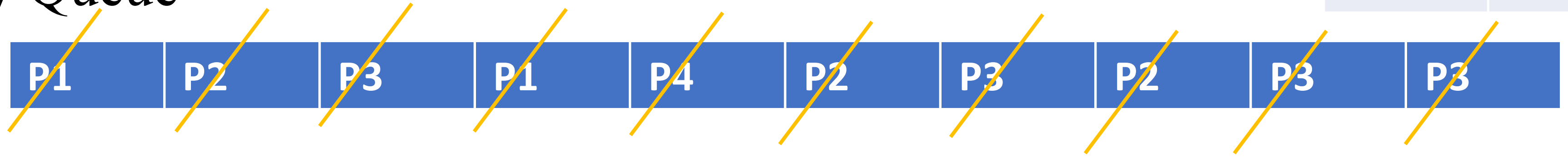
Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
P1	0	3	3	3	0
P2	1	5	9	8	3
P3	2	7	16	14	7
P4	3	1	4	1	0
∑				26	10

Average TAT = ∑TAT/no. of process = 26/4 = 6.5 ms
Average WT = ∑WT/no. of process = 10/4 = 2.5 ms

Process	Arrival Time(AT)	CPU Time(CT)
P1	0	3
P2	1	5
P3	2	7
P4	3	1

According to RR

- Ready Queue



- Ghantt Chart

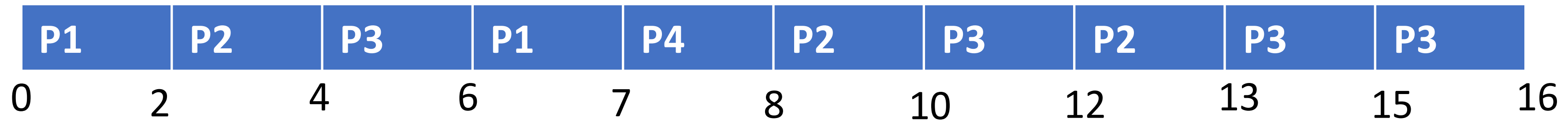


Fig: Ghantt chart according to RR

Proces s	Arrival Time(AT)	CPU Time(CT)
P1	0	3
P2	1	5
P3	2	7
P4	3	1

According to RR

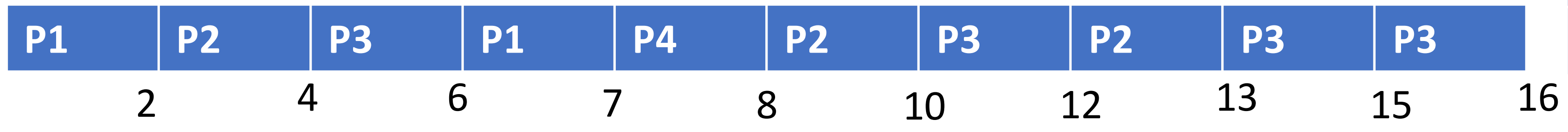


Fig: Ghantt chart according to RR

Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
P1	0	3	7	7	4
P2	1	5	13	12	7
P3	2	7	16	14	7
P4	3	1	8	5	4
∑				38	22

Average TAT = ∑TAT/no. of process = 38/4 = 9.5 ms

Average WT = ∑WT/no. of process = 22/4 = 5.5 ms

NUMERICALS

3. Schedule the following set of process according to:
    i. FCFS
    ii. SJF
    iii. SRTN
    iv. RR(time quantum = 2 ms)

    And calculate **average waiting time** and **average turnaround time**.

Process	Arrival Time(ms)	CPU Time(ms)
P0	0	3
P1	2	6
P2	4	4
P3	6	5
P4	8	2

Process	Arrival Time(AT)	CPU Time(CT)
P0	0	3
P1	2	6
P2	4	4
P3	6	5
P4	8	2

According to FCFS

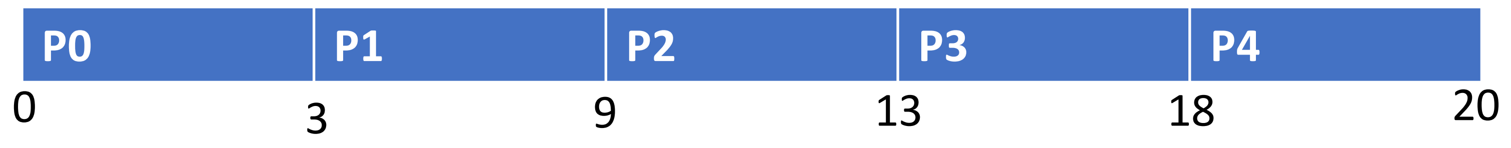


Fig: Ghantt chart according to FCFS

Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
P0	0	3	3	3	0
P1	2	6	9	7	1
P2	4	4	13	9	5
P3	6	5	18	12	7
P4	8	2	20	12	10
∑				43	23

Average TAT = ∑TAT/no. of process = 43/5 = 8.6 ms
Average WT = ∑WT/no. of process = 23/5 = 4.6 ms

According to SJF

Process	Arrival Time(AT)	CPU Time(CT)
P0	0	3
P1	2	6
P2	4	4
P3	6	5
P4	8	2

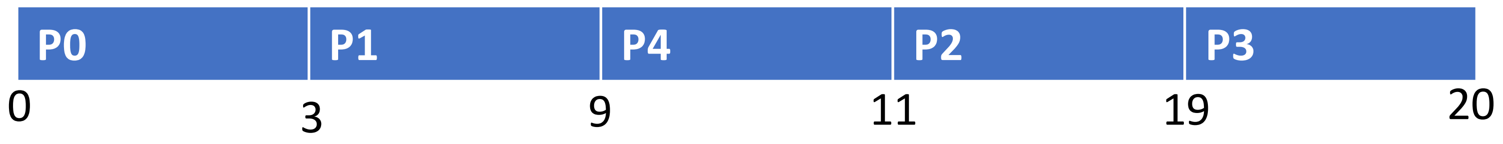


Fig: Ghantt chart according to SJF

Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
P0	0	3	3	3	0
P1	2	6	9	7	1
P2	4	4	15	11	7
P3	6	5	20	14	9
P4	8	2	11	3	1
∑				38	18

Average TAT = ∑TAT/no. of process = 38/5 = 7.6 ms
Average WT = ∑WT/no. of process = 18/5 = 3.6 ms

Process	Arrival Time(AT)	CPU Time(CT)
P0	0	3
P1	2	6
P2	4	4
P3	6	5
P4	8	2

According to SRTN

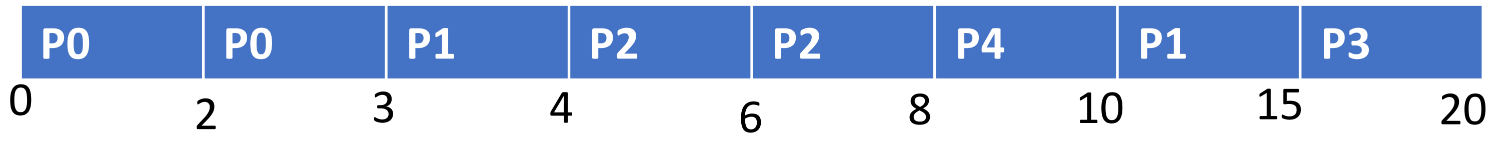


Fig: Ghantt chart according to SRTN

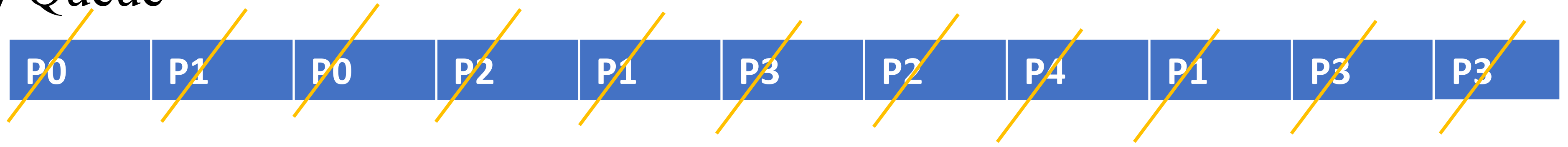
Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
P0	0	3	3	3	0
P1	2	6	15	13	7
P2	4	4	8	4	0
P3	6	5	20	14	9
P4	8	2	10	2	0
∑				36	16

NOTE: In SRTN, If 2 processes have same Burst Time, then allocate CPU according to FCFS basics.

Average TAT = ∑TAT/no. of process = 36/5 = 7.2 ms
Average WT = ∑WT/no. of process = 16/5 = 3.2 ms

According to RR (Quantum time = 2ms)

- Ready Queue



- Ghantt Chart

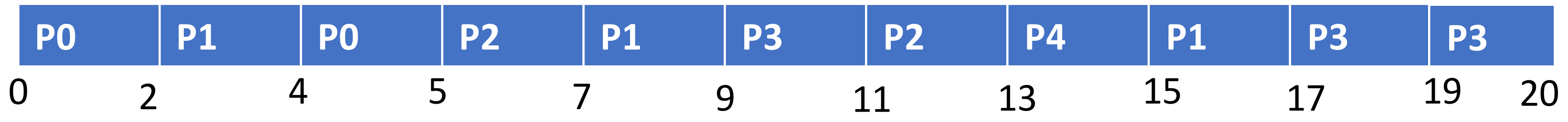


Fig: Ghantt chart according to RR

Process	Arrival Time(AT)	CPU Time(CT)
P0	0	3
P1	2	6
P2	4	4
P3	6	5
P4	8	2

According to RR

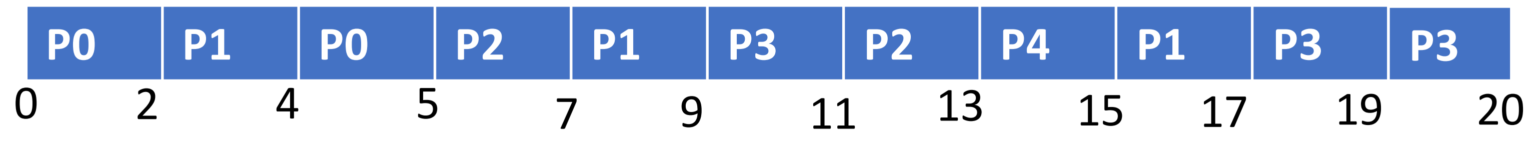


Fig: Ghantt chart according to RR

Process	Arrival Time(AT)	CPU Time/ Burst Time	Completion Time(CT)	TAT= CT-AT	WT= TAT-BT
P0	0	3	5	5	2
P1	2	6	17	15	9
P2	4	4	13	9	5
P3	6	5	20	14	9
P4	8	2	15	7	5
∑				50	30

Average TAT = ∑TAT/no. of process = 50/5 = 10 ms

Average WT = ∑WT/no. of process = 30/5 = 6 ms

Assignment

- Assignment- I
- Assignment -II
- Solve remaining(2 questions) & 3(numerical PAST QUESTIONS)

Submission date:- Tuesday

NUMERICALS

4. Let us consider following processes with given arrival time and length of cpu burst given in millisecond(s).

i. Draw the Gantt chart showing the execution of FCFS, SRTN and RR(Quantum =2 ms)

ii. Calculate **average waiting time** and **average turn around time** & which algorithm results in maximum turn around time

Process	Arrival Time(ms)	CPU Time(ms)
P1	0	4
P2	1	5
P3	2	2
P4	3	1
P5	4	6
P6	6	3

NUMERICALS

5. Let us consider following processes with given arrival time and length of cpu burst given in millisecond(s).

i. Calculate **average waiting time** and **average turn around time** for FCFS, SJF, SRTN, RR(Q = 3ms) and HRRN.

Process	Arrival Time(ms)	CPU Time(ms)
P1	0	7
P2	1	5
P3	2	3
P4	3	4
P5	4	6

Completely Fair Scheduler(CFS) used in Linux

- It is the default scheduling process since version 2.6.23.
- Elegant handling of I/O and CPU bound process
- As the name suggests, it fairly or equally divides the CPU time among all the processes.
- Before understanding the CFS let's look at the **Ideal Fair Scheduling (IFS)** of N processes.
- If there are N processes in the ready queue then each process receives (100/N)% of CPU time according to **IFS.**

Completely Fair Scheduler(CFS) used in Linux

- Let’s take four process and their burst time as shown below waiting in the ready queue for the execution.

Process	Burst Time (in ms)
A	8
B	4
C	16
D	4

Completely Fair Scheduler(CFS) used in Linux

- Take a time quantum of say 4ms. **Initially, there are four processes waiting in the ready queue to be executed**, and according to **Ideal Fair Scheduling**, each process gets equally fair time for its execution (Time quantum/N).
- So, 4/4 = 1, each process gets 1ms to execute in the first quantum. After the completion of four quanta, Process B and D are completely executed, and the remaining are A and C, which have already executed for 4ms, with their remaining times being A = 4ms and C = 12ms.
- In the fifth quantum of time, A and C will execute (4/2 = 2ms as there are only two processes remaining). This is Ideal Fair Scheduling, in which each process gets an equal share of the time quantum, no matter what priority it has.

Completely Fair Scheduler(CFS) used in Linux

Processes	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	
A	1	2	3	4	6	8	-	-	
B	1	2	3	4	-	-	-	-	
C	1	2	3	4	6	8	12	16	
D	1	2	3	4	-	-	-	-	

Process	Burst Time (in ms)
A	8
B	4
C	16
D	4

CFS is similar as ideal-based scheduling instead it priorities each process according to their virtual runnable time.

Virtual Runtimes

- With each runnable process, is included a virtual runtime(vruntime)
  - At every scheduling point, if process has run for t ms, then (vruntime +=t)
  - vruntime for a process therefore monotonically increases

The CFS idea

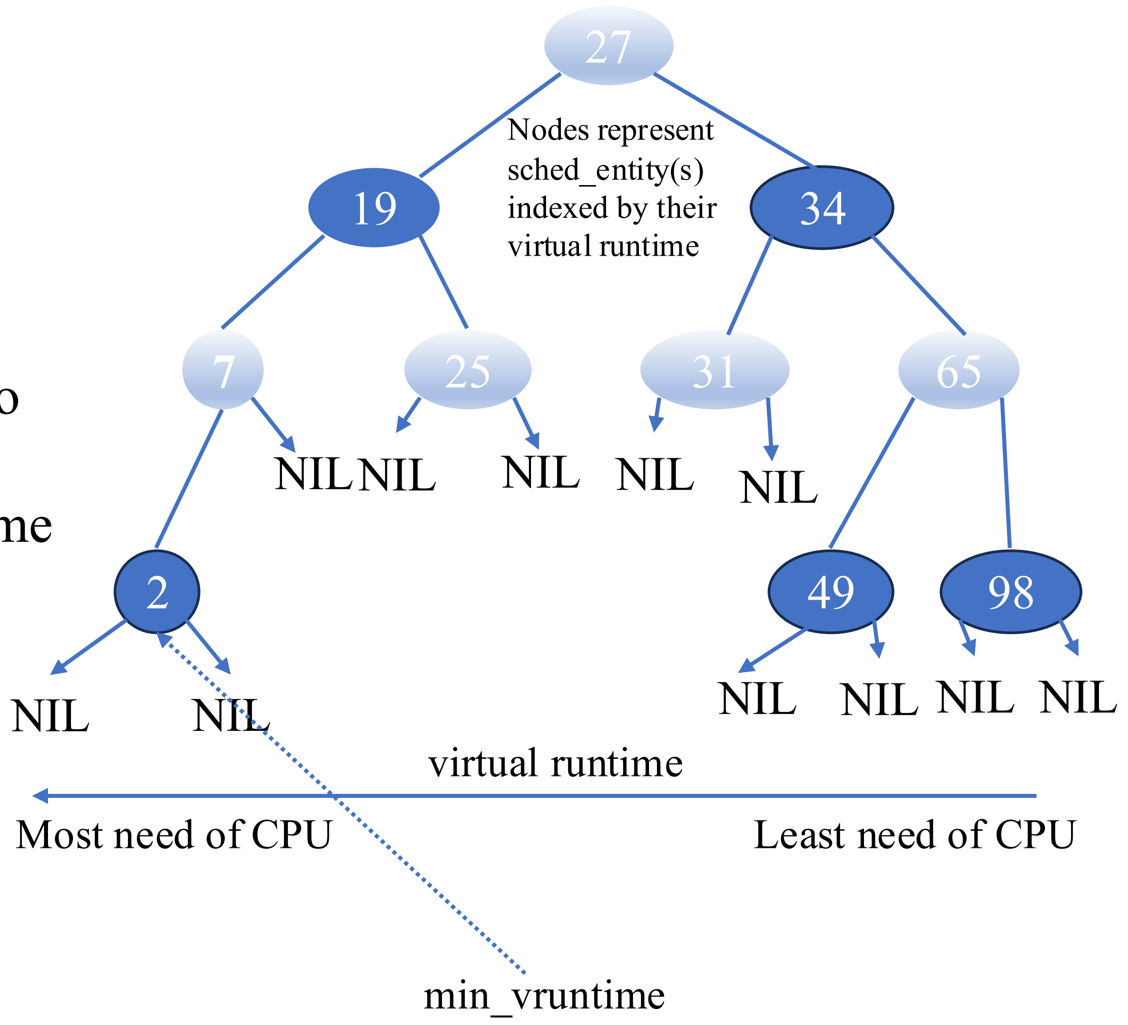
- When timer interrupt occurs
  - Choose the task with the lowest vruntime(min_vruntime)
  - Compute its dynamic timeslice
  - Program the high resolution timer with its timeslice

- The process begins to execute in the CPU

- When interrupt occurs again
  - Context switch if there is another task with a smaller runtime

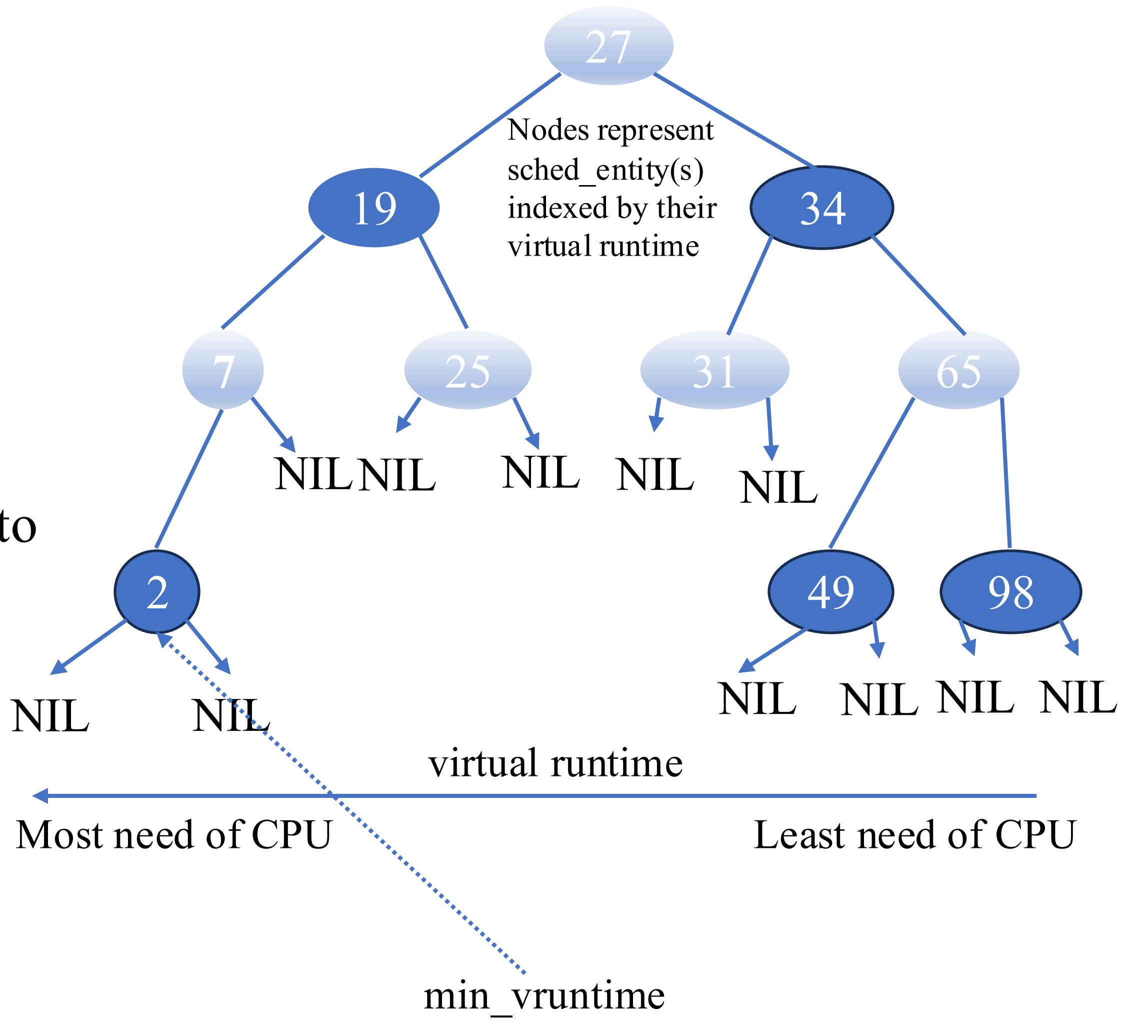
Picking the next task to run

- CFS uses a **red-black tree.**
  - ➢ Each node in the tree represents a runnable task
  - ➢ Nodes ordered according to their vruntime
  - ➢ Nodes on the left have lower vruntime compared to nodes on the right of the tree
  - ➢ The left most node is the task with the least vruntime
    - • This is cached in min_vruntime.



Picking the next task to run

- At a context switch,
  - ➢ Pick the left most node of the tree
    - This has the lowest runtime
    - It is cached in min_vruntime. Therefore accessed in O(1).
  - ➢ If the previous process is runnable, it is inserted into the tree depending on its new vruntime. Done in O(log(n))
    - Tasks move from left to right of tree after its execution completes… starvation avoided



Why Red Black Tree?

- Self Balancing
  - No path in the tree will be twice as long as any other path
- All operations are O(log(n))
  - Thus inserting/ deleting tasks from the tree is quick and efficient

Priorities and CFS

- Priority(due to nice values) used to weigh the vruntime

- If process has run for t ms, then
  - vruntime += t * (weight based on nice of process)

- A low priority implies time moves at a faster rate compared to that of a high priority task

How to calculate dynamic timeslice?

- timeslice=(weight of task/ total weight of all tasks)×target latency
- **Target latency**: The maximum time all runnable tasks should get to run once. (e.g., 20ms)
- So, if there are 4 equal-weight tasks, each gets ≈ 5ms.

I/O and CPU Bound processes

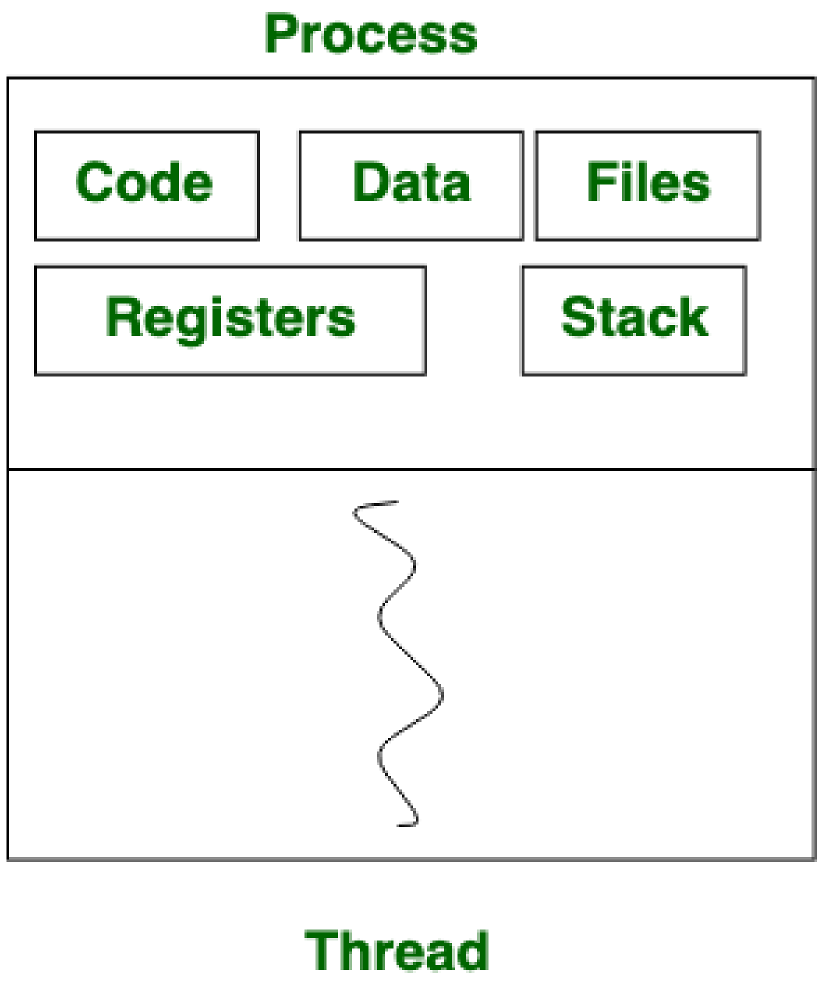
- What we need,
  - I/O bound should get higher priority and get a longer time to execute compared to CPU bound
  - CFS achieves this efficiently
    - I/O Bound processes have small CPU bursts therefore will have a low vruntime. They should appear towards the left of the tree... Thus are given higher priorities
    - I/O Bound processes will typically have larger time slices, because they have smaller vruntime

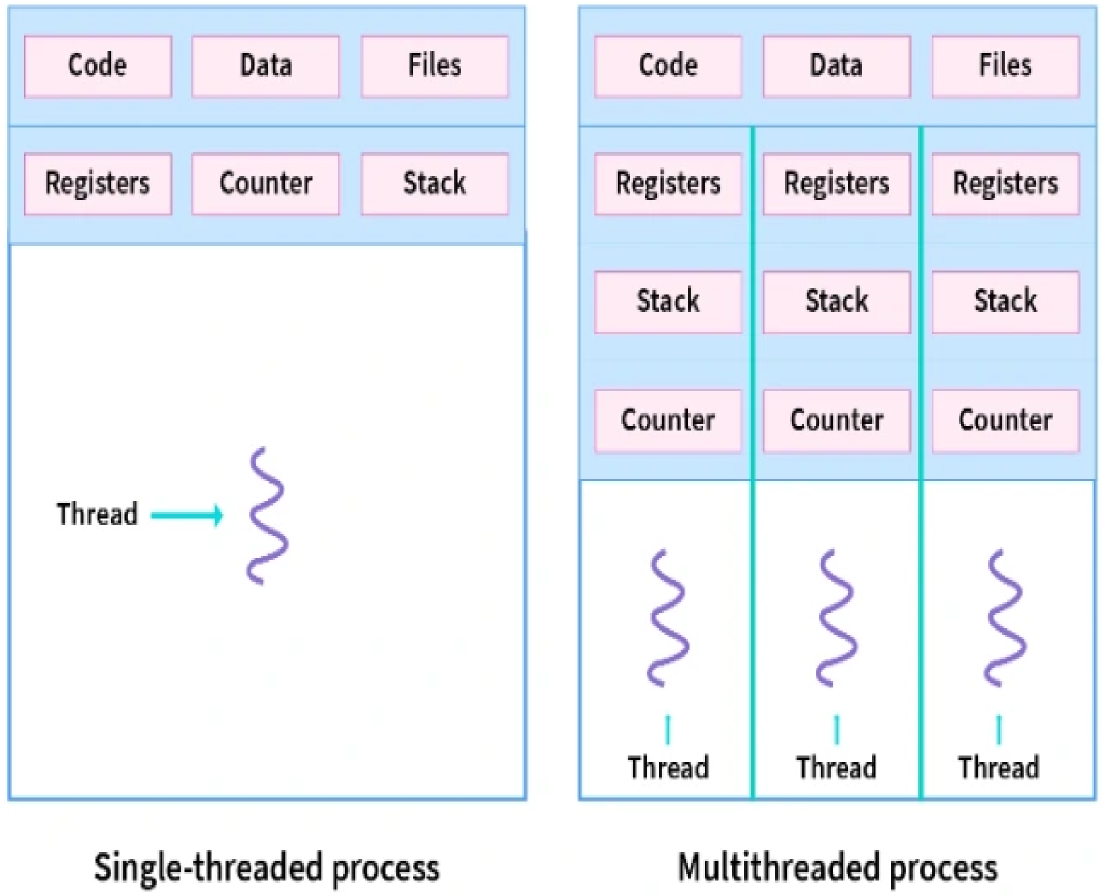
CPU Bound Vs I/O Bound

S.N.	CPU Bound	I/O Bound
1.	CPU Bound process are the process for which the time to complete the task is correlated to the speed of the central processor.	I/O Bound process are the process for which the time to complete the task is dependent on the period specifically waiting for I/O operations to be completed.
2.	CPU Bound process is faster process than I/O Bound process.	I/O Bound process is slower than CPU Bound process.
3.	CPU Bound process spends most of the time using CPU.	I/O Bound process spends most of the time using I/O.
4.	It takes longer CPU Bursts.	It takes shorter CPU Bursts.
5.	E.g: Video Editing, Gaming e.t.c.	E.g: Web browser, Text editor, Copying files, e.t.c

Thread

- A thread is a single sequence stream within in a process.
- Because threads have some of the properties of processes, they are sometimes called **lightweight processes**.
- In many respect, threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.
- Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or terminated).
- A thread consists of a program counter (PC), a register set, and a stack space.
- Threads are not independent of one other like processes. Threads shares address space, program code, global variables, OS resources with other thread.


Process
Code
Data
Files
Registers
Stack
Thread


Code
Data
Files
Registers
Counter
Stack
Thread
Single-threaded process
Code
Data
Files
Registers
Registers
Registers
Stack
Stack
Stack
Counter
Counter
Counter
Thread
Thread
Thread
Multithreaded process

Process Vs. Thread

PROCESS	THREAD
Doesn’t share memory (loosely coupled).	Shares memories and files (tightly coupled)
Creation is time consuming.	Fast
Execution is slow.	Fast
More time to terminate.	Less time
More time to switch between processes.	Less time
System calls are required for communication.	Not required
More resources are required.	Fewer resources are required
Not suitable for parallelism.	Suitable

Multithreading

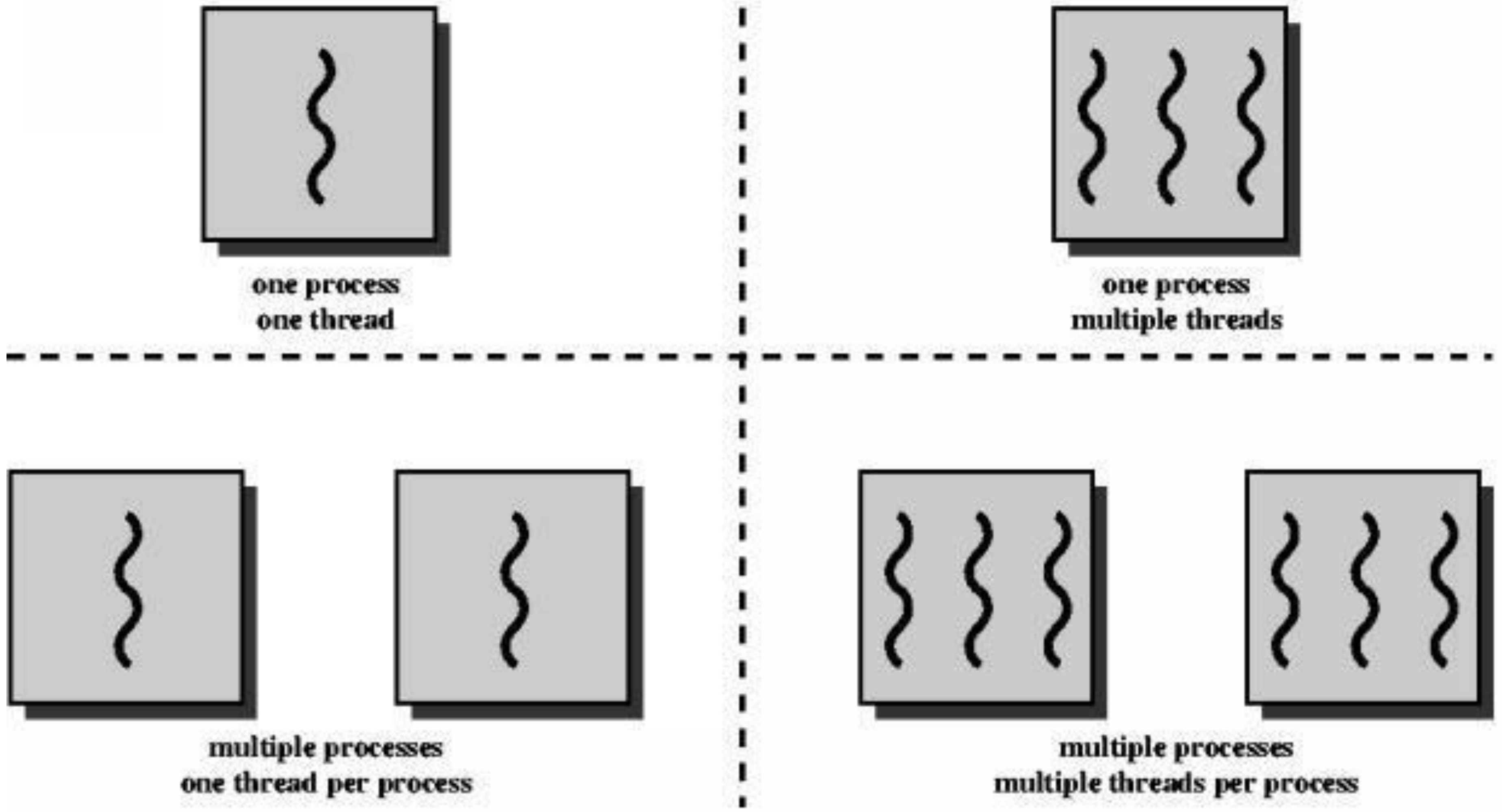
- The use of multiple threads in a single program, all running at the same time and performing different tasks is known as multithreading.

- Multithreading is a type of execution model that allows multiple threads to exist within the context of a process such that they execute independently but share their process resources.

- It enables the processing of multiple threads at one time, rather than multiple processes.

Based on functionality, threads are put under 4 categories:

- Single Process, Single Thread (MS-DOS)
- Single Process, Multiple Threads (JAVA Runtime)
- Multiple Process, Single Thread per process (Unix)
- Multiple Process, Multiple Threads (Solaris, Unix)


one process
one thread
one process
multiple threads
multiple processes
one thread per process
multiple processes
multiple threads per process

Types of Thread

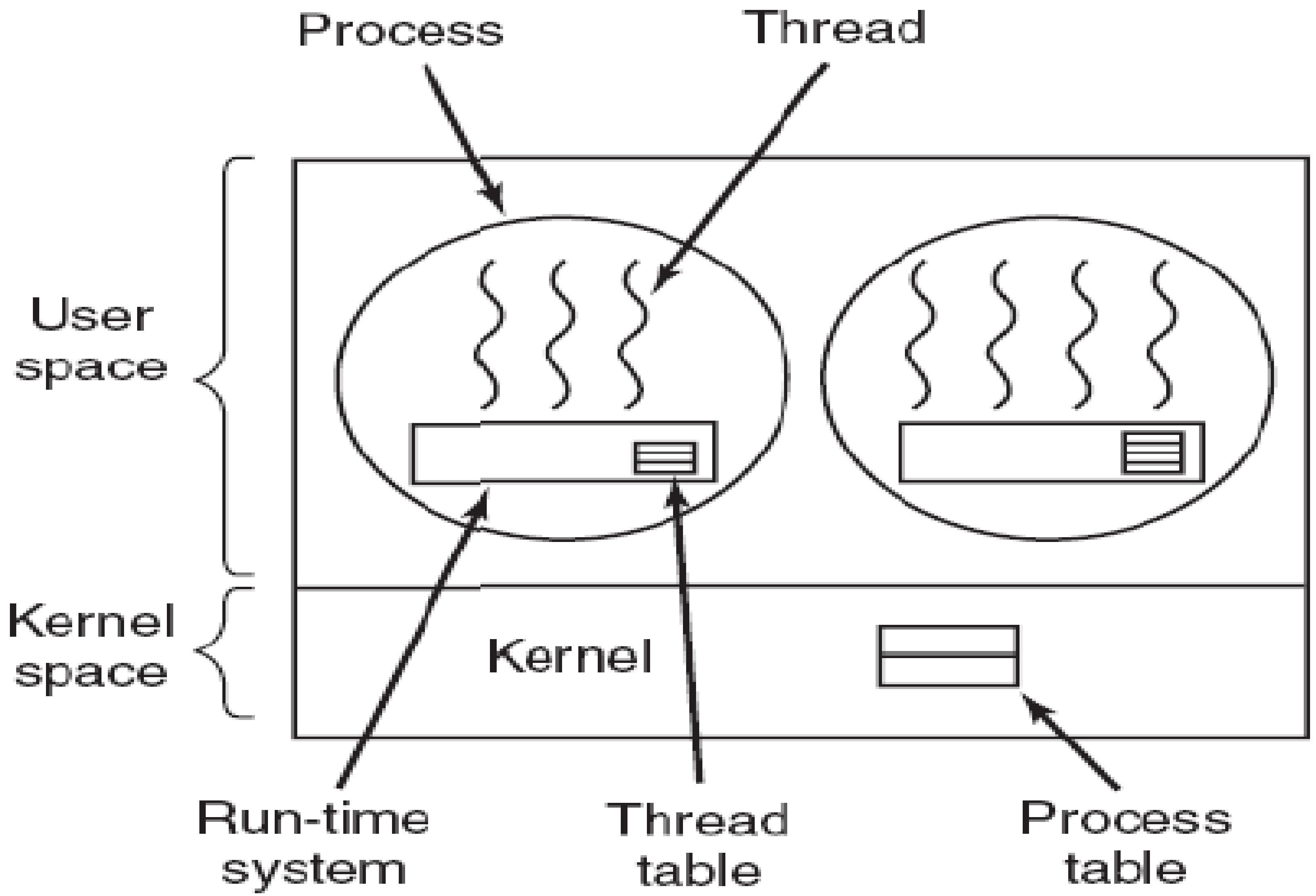
- Thread are implemented in following two ways:
  - User Level Threads − User managed threads.
  - Kernel Level Threads − Operating System managed threads acting on kernel, an operating system core.

User Level Threads

- User thread are supported at user level.
- In this case, the thread management kernel is not aware of the existence of threads.
- The **thread library** contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts.
- In fact, the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.

Advantages:

- The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads.
- User-level thread does not require modification to operating systems.
- **Simple Representation:** Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- **Simple Management:** This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- **Fast and Efficient:** Thread switching is not much more expensive than a procedure call.


Process
Thread
User space
Kernel space
Kernel
Run-time system
Thread table
Process table

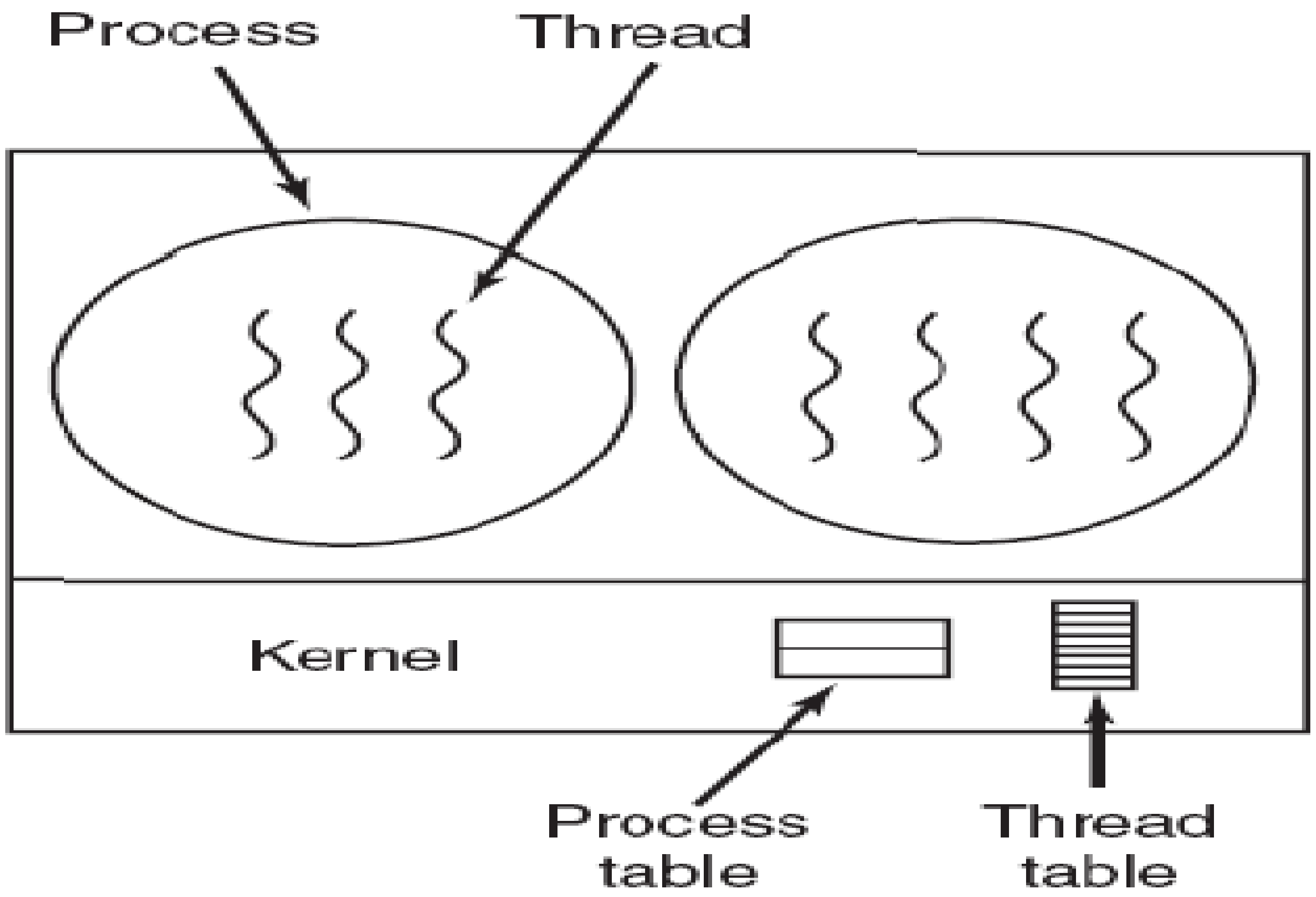
Disadvantages

- There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.

- User-level thread requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will be blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

Kernel-Level Threads

- In this method, the kernel knows about and manages the threads.
- Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system.
- Scheduling by the Kernel is done on a thread basis.
- The Kernel performs thread creation, scheduling and management in Kernel space.
- Kernel threads are generally slower to create and manage than the user threads.


Process
Thread
Kernel
Process table
Thread table

Advantages

- Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.

Disadvantages:

- The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.

- Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads. As a result, there is significant overhead and increased in kernel complexity.

User Level Vs. Kernel Level Thread

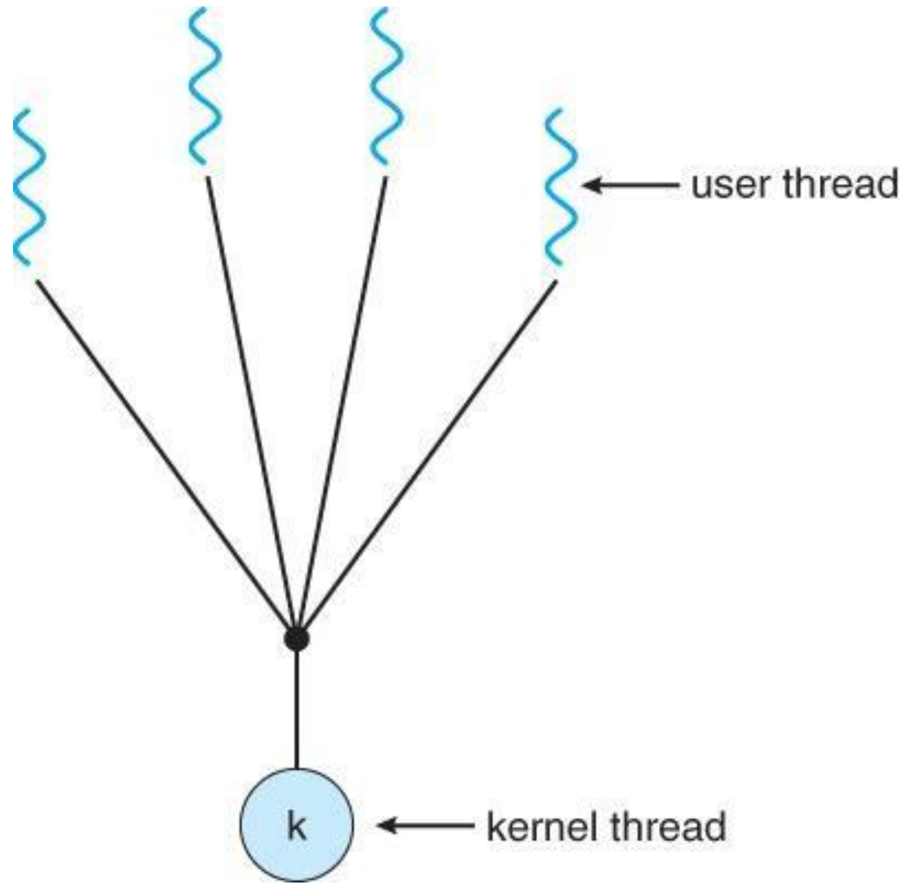
User-Level Threads	Kernel-Level Thread
User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Multithreading Models

- Many-To-One Model
- One-To-One Model
- Many-To-Many Model

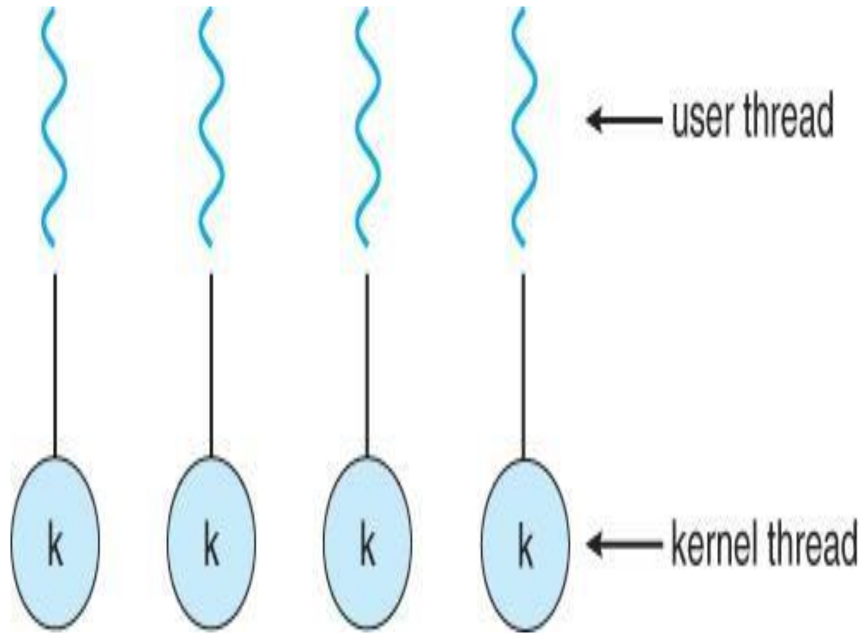
Many-to-one model

- Many user-level threads are all mapped onto a single kernel thread.
- Thread management is done by the thread library in user space, so it is efficient.
- However, if a blocking system call is made, then the entire process blocks.
- Because only one thread can access the kernel at the same time, multiple threads are unable to run in parallel on microprocessors.



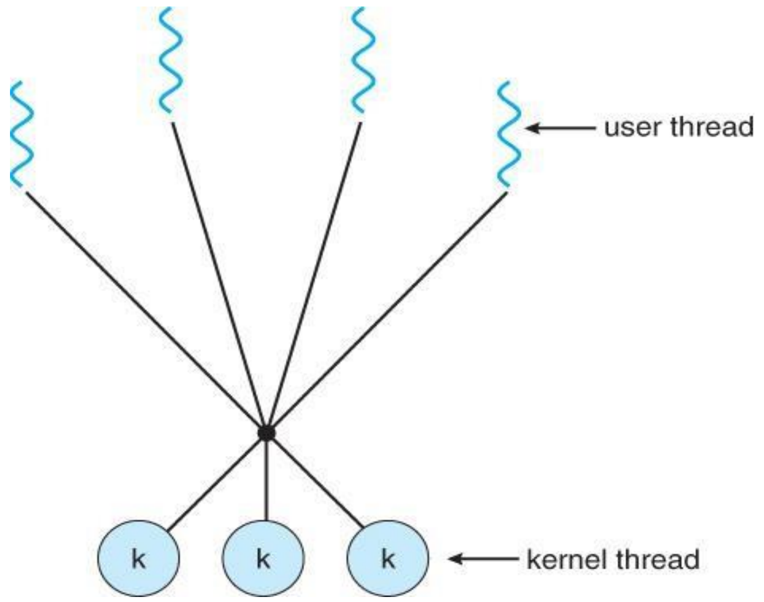
One-to-one Model

- Maps each user thread to a kernel thread
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However, the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- Most implementations of this model place a limit on how many threads can be created.
- **Linux** and **Windows** from 95 to XP implement the one-to-one model for threads.

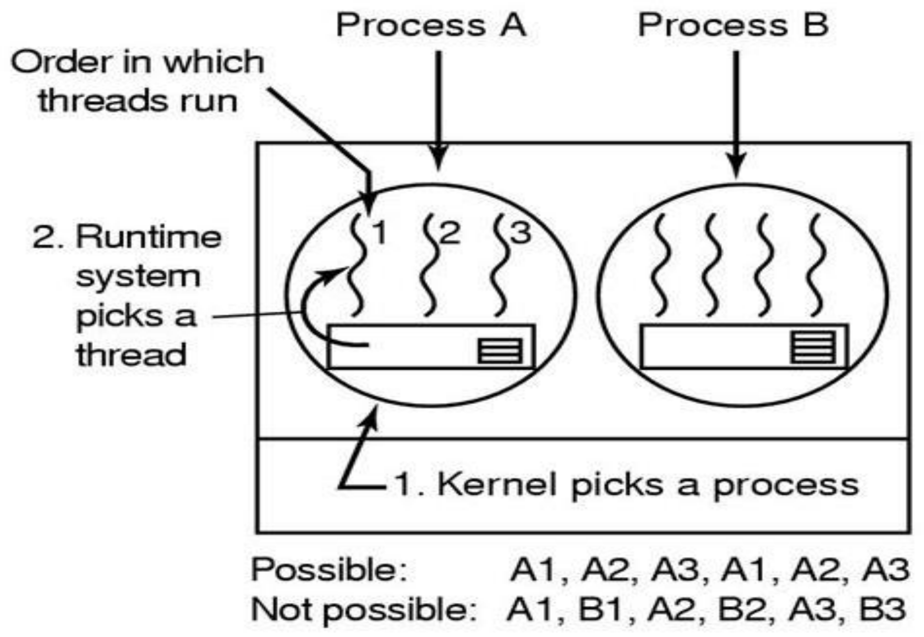


Many-to-many Model

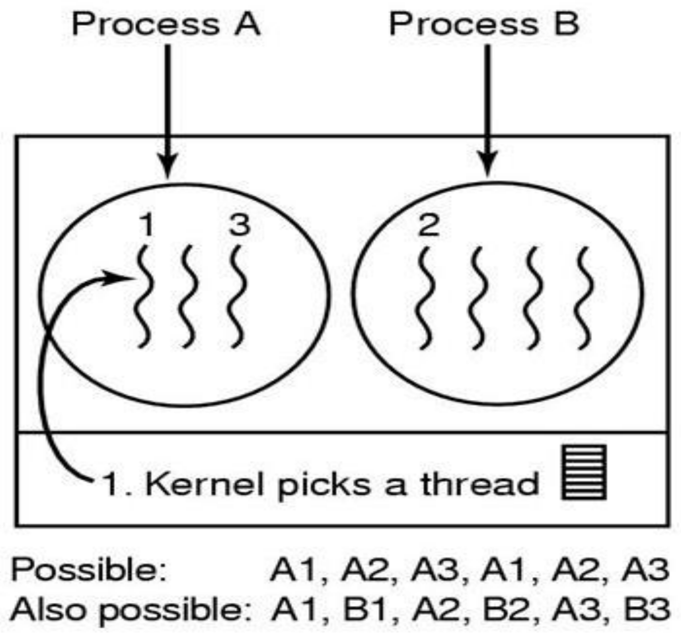
- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.



Thread Scheduling


Process A
Process B
Order in which threads run
2. Runtime system picks a thread
1. Kernel picks a process
1 2 3
Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3

(a)


Process A
Process B
1 3
2
1. Kernel picks a thread
Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

(b)

Fig: (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5msec per CPU burst. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

Thread Scheduling

- When several processes each have multiple threads, we have two levels of parallelism present: processes and threads. Scheduling in such systems differs substantially depending on whether user-level threads or kernel-level threads (or both) are supported.
- Let us consider user-level threads first. Since the kernel is not aware of the existence of threads, it operates as it always does, picking a process, say, A , and giving A control for its quantum. The thread scheduler inside A decides which thread to run, say A1 . Since there are no clock interrupts to multiprogram threads, this thread may continue running as long as it wants to. If it uses up the process' entire quantum, the kernel will select another process to run.

Thread Scheduling

- When the process A finally runs again, thread A1 will resume running. It will continue to consume all of A ’s time until it is finished. However, its antisocial behavior will not affect other processes. They will get whatever the scheduler considers their appropriate share, no matter what is going on inside process A .
- Now consider the case that A ’s threads have relatively little work to do per CPU burst, for example, 5 msec of work within a 50-msec quantum. Consequently, each one runs for a little while, then yields the CPU back to the thread scheduler. This might lead to the sequence A1 , A2 , A3 , A1 , A2 , A3 , A1 , A2 , A3 , A1 , before the kernel switches to process B . This situation is illustrated in Fig.(a).
- The scheduling algorithm used by the run-time system can be any of the ones described above. In practice, **round-robin scheduling** and **priority scheduling** are most common. The only constraint is the absence of a clock to interrupt a thread that has run too long.

Thread Scheduling

- Now consider the situation with kernel-level threads. Here the kernel picks a particular thread to run. It does not have to take into account which process the thread belongs to, but it can if it wants to. The thread is given a quantum and is forceably suspended if it exceeds the quantum. With a 50-msec quantum but threads that block after 5 msec, the thread order for some period of 30 msec might be A1 , B1 , A2 , B2 , A3 , B3 , something not possible with these parameters and user-level threads. This situation is partially depicted in Fig. (b).
- A major difference between user-level threads, and kernel-level threads is the performance. Doing a thread switch with user-level threads takes a handful of machine instructions. Wit kernel-level threads it requires a full context switch, changing the memory map, and invalidating the cache, which is several orders of magnitude slower. On the other hand, with kernel-level threads, having a thread block on I/O does not suspend the entire process as it does with user- level threads.