MIT-WPU

।। विश्वशान्तिर्ध्रुवं ध्रुवा ।।

Dr. Vishwanath Karad

MIT WORLD PEACE UNIVERSITY | PUNE

TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

Scheduling

Module 2

Process Management

- **Process:** Concept of a Process, Process States, Process Control - creation, new program execution, termination. Interposes communication(IPC). Examples of IPC.

- **Threads:** Differences between Threads and Processes. Concept of Threads, Concurrency. Multi- threading, Types of Threads. POSIX Threads functions.

- **Scheduling:** Concept of Scheduler, Scheduling Algorithms: FCFS, SJF, SRTN, Priority, Round Robin.

Scheduling

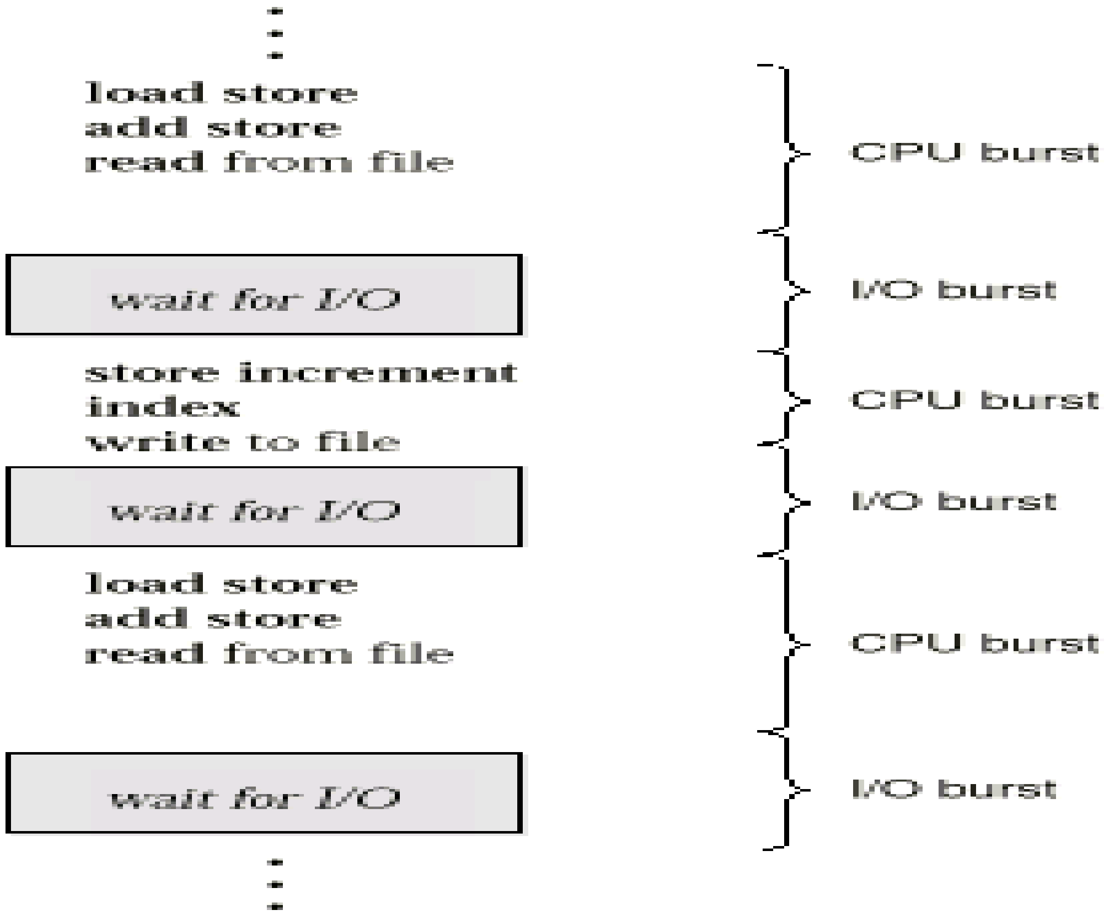
- **Scheduling:** Concept of Scheduler, Scheduling Algorithms: FCFS, SJF, SRTN, Priority, Round Robin.
-

Scheduling

- Maximum CPU utilization obtained with multiprogramming

- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.

Alternating Sequence of CPU And I/O Bursts



CPU / Process Scheduler

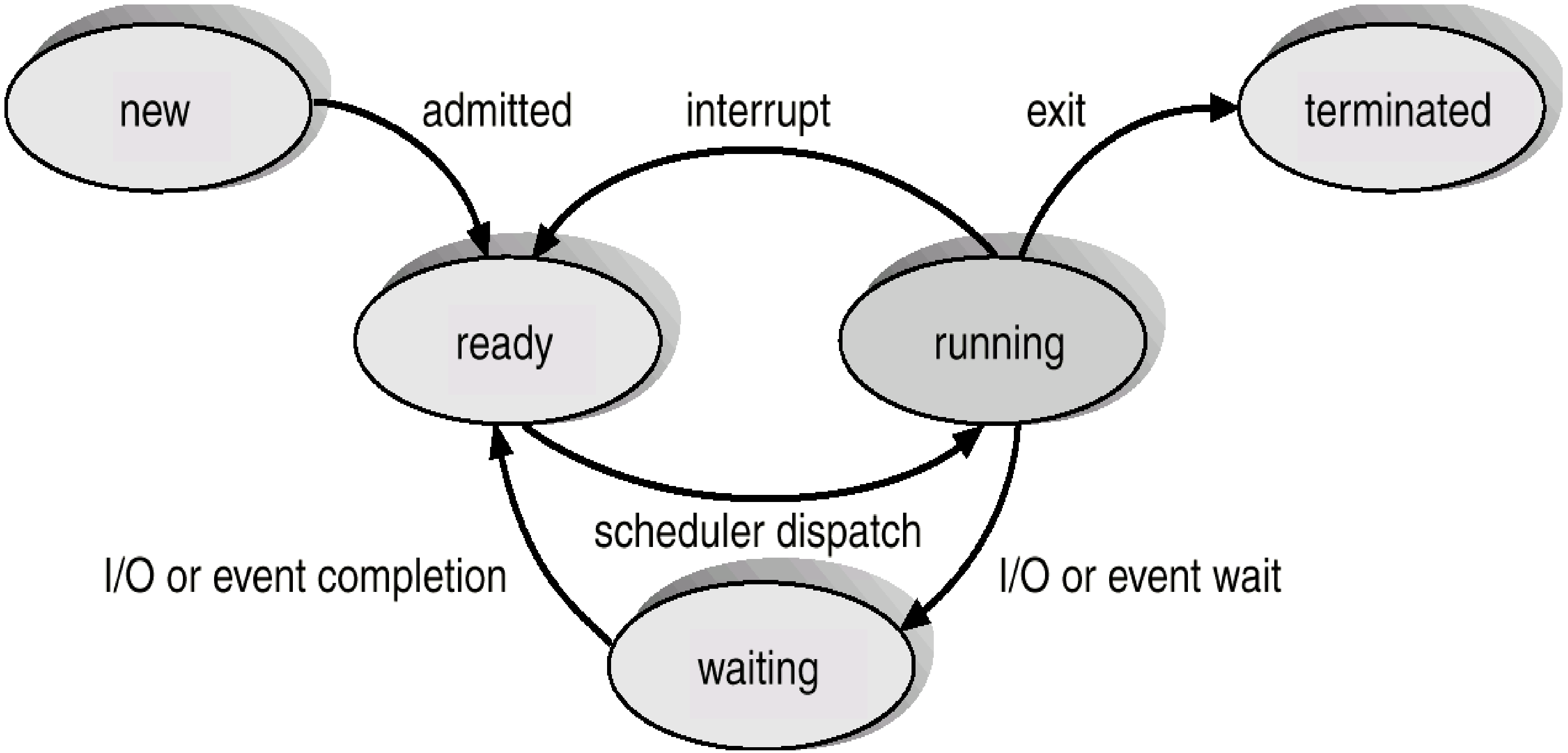
- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

- Scheduling may be *preemptive or non-preemptive*
- ***Non-preemptive*** – If CPU allocated to a process, it keeps the CPU until it releases it by terminating or switching to waiting state
- ***Preemptive*** - If CPU allocated to a process, it may be released if high priority process needs the CPU

Scheduler

- 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 state
  4. Terminates
- Scheduling under 1 and 4 is *non-preemptive*
- All other scheduling is *preemptive*

Diagram for Process States



Scheduling Algorithms

They deal with the problem of deciding which of the processes in ready queue is to be allocated the CPU

Algorithm compared based on following criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – number of processes completed or amount of work done per unit time
- **Turnaround time** – time of submission of a process to the time of completion
- **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

Optimization Criteria

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

Types of process schedulers / Scheduling categories

Scheduling is broken down into three categories:

1. Long term scheduling:

- Is performed when a new process is created.
- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue.
- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*.

Types of process schedulers / Scheduling categories

2. Medium term scheduling:

- Part of the swapping function
- Swapping-in decisions are taken by medium term scheduler
- Based on the need to manage the degree of multiprogramming

Types of process schedulers / Scheduling categories

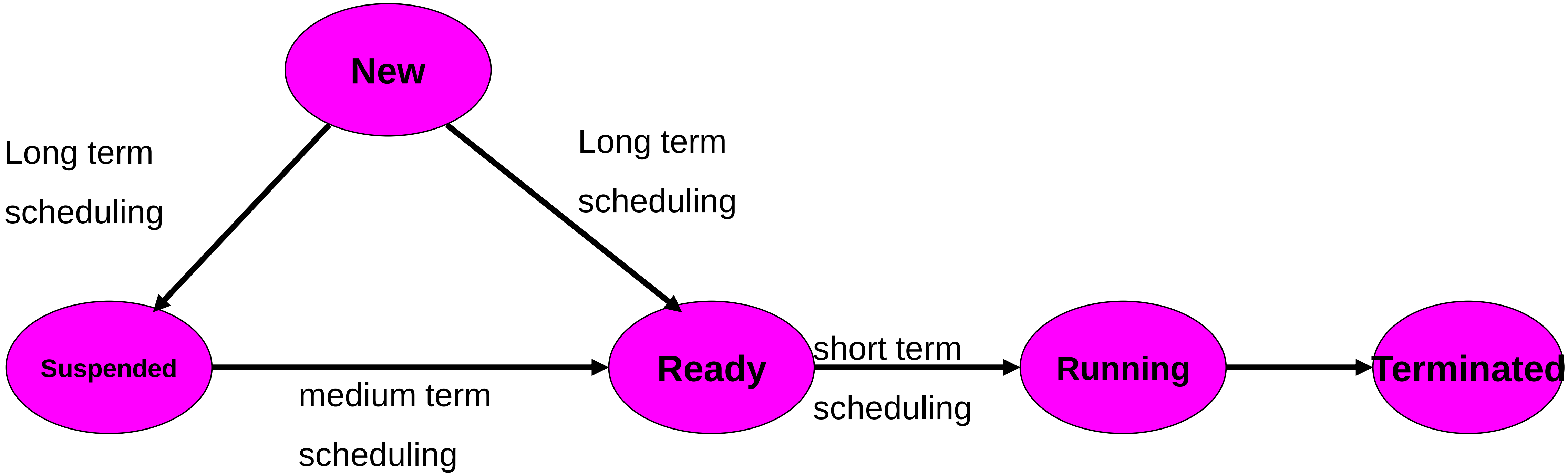
3.Short term scheduling:

- Determines which ready process will be assigned the CPU when it next becomes available and actually assign the CPU to this process **.**

- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU.

- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast).

Types of process schedulers / Scheduling categories



Scheduling Algorithms

- **FCFS (First Come First Serve)**
- **SJF (Shortest Job First)**
- **Priority scheduling**
- **Round Robin scheduling**

FCFS Scheduling: Characteristics

Selection Function: max(w), selects the process which is waiting in the ready queue for maximum time.

Decision Mode : Non_preemptive

Throughput: Not emphasized

Response Time: May be high, especially if there is a large variance in the process execution times.

Overhead: Minimum

Effect on Processes: Penalizes short processes

Starvation: No

- **Completion Time**

  Time at which process completes its execution.

- **Turn Around Time**

  Time Difference between completion time and arrival time.

  Turn Around Time = Completion Time – Arrival Time

- **Waiting Time(W.T)**

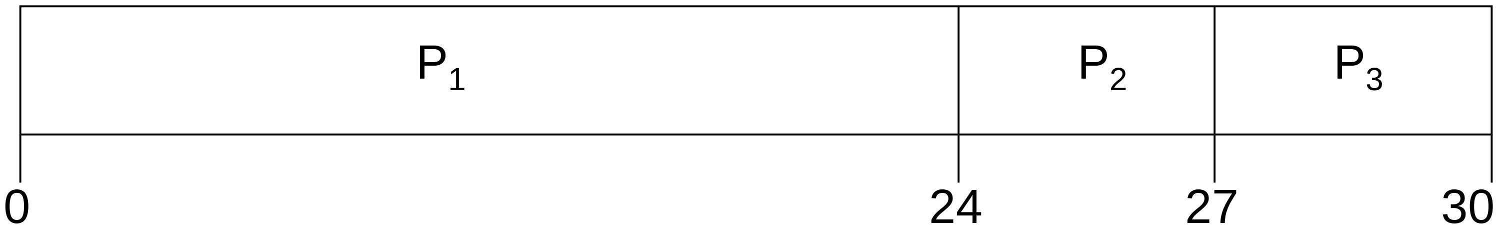
  Time Difference between turn around time and burst time.

  Waiting Time = Turn Around Time – Burst Time

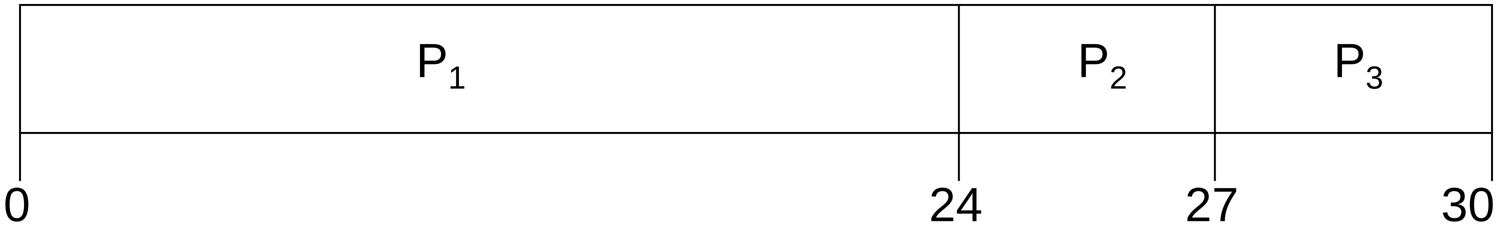
Process	**Burst Time**
P1	24
P2	3
P3	3

- Suppose that the processes arrive in the order: *P1* , *P2* , *P3*

The Gantt Chart for the schedule is:



The Gantt Chart for the schedule is:

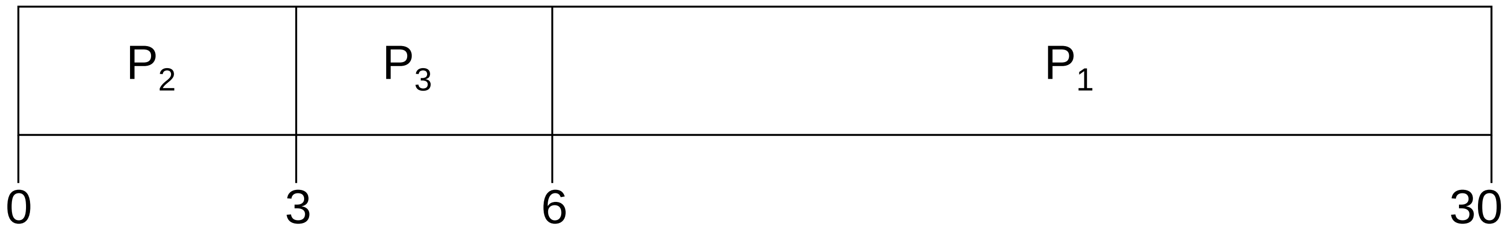


Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
Average waiting time: (0 + 24 + 27)/3 = 17

Turnaround time for *P1* = 24; *P2* = 27; *P3* = 30
Average turnaround time : (24 + 27 + 30)/3 = 27

FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order : *P2* , *P3* , *P1*



FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order : *P2* , *P3* , *P1*

Waiting time for *P1* = 6; *P2* = 0; *P3* = 3
Average waiting time: (6 + 0 + 3)/3 = 3

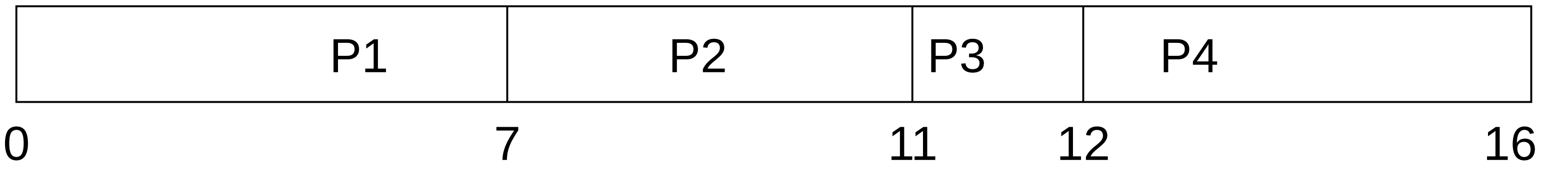
Turnaround time for *P1* = 30; *P2* = 3; *P3* = 6
Average turnaround time : (30 + 3 + 6)/3 = 13

Example-2 of FCFS

Process	Arrival Time	Burst Time
P1	0.0	3
P2	2.0	6
P3	4.0	4
P4	6.0	5
P5	8.0	2

Example-2 of FCFS

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4



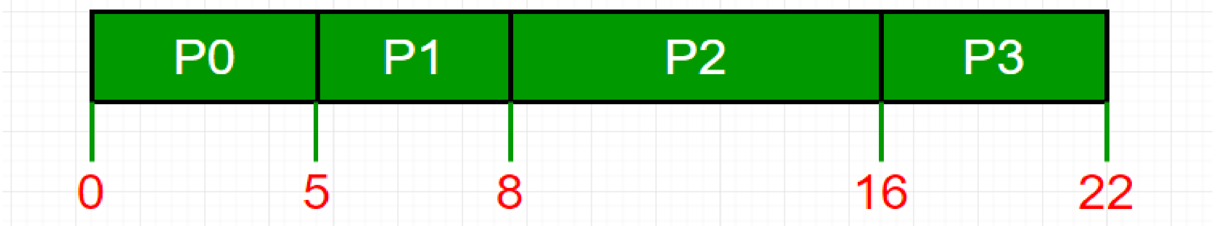
Average waiting time = (0 + 5 + 7 + 7)/4 = 4.75
Average Turnaround Time = (7+9+8+11)/4=8.75

First Come First Serve (FCFS) (Non Pre-emptive)

Processes	Burst time	Arrival Time	Service Time
P0	5	0	0
P1	3	1	5
P2	8	2	8
P3	6	3	16

- **Service Time :** Service time means amount of time after which a process can start execution.

It is summation of burst time of previous processes (Processes that came before)



First Come First Serve (FCFS) (Non Pre-emptive)

Processes	Burst time	Arrival Time	Service Time
P0	5	0	0
P1	3	1	5
P2	8	2	8
P3	6	3	16

- **To find waiting time:**

  Time taken by all processes before the current process to be started

  (i.e. burst time of all previous processes) – arrival time of current process

  wait_time[i] = (bt[0] + bt[1] +…… bt[i-1]) – arrival_time[i]

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0 + 4 + 6 + 13) / 4 = 5.75

First Come First Serve (FCFS) (Non Pre-emptive)

Implementation

1) Input the processes along with their burst time (bt) and arrival time (at).

2) Find **waiting time (wt)** for all processes. i.e. for a given process i:

   wt[i] = (bt[0] + bt[1] + ... + bt[i-1]) - at[i] .

3) Now find **turnaround time** = waiting_time + burst_time for all processes.

4) Find **average waiting time** = total_waiting_time / no_of_processes.

5) Similarly, find **average turnaround time** = total_turn_around_time / no_of_processes.

Shortest-Job-First (SJF) Scheduling

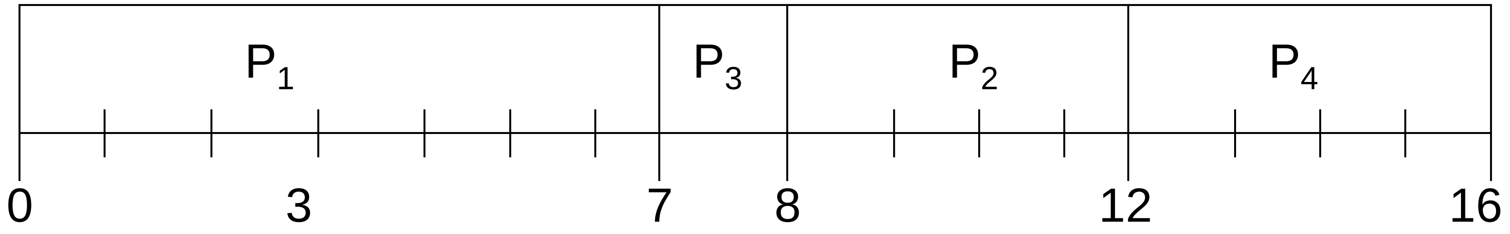
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - **Nonpreemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst
  - **Preemptive** – If a new process arrives with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

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4



- **Average waiting time = (0 + 6 + 3 + 7)/4 = 4**
- **Average Turnaround Time = (7+10+4+11)/4=8**

Shortest Job First Preemptive or Shortest Remaining Time

- It is a preemptive version of SJF. In this policy, scheduler always chooses the process that has the **shortest expected remaining processing time.**

- When a new process arrives in the ready queue, it may in fact have a shorter remaining time than the currently running process.

- Accordingly, the scheduler may preempt whenever a new process becomes ready.

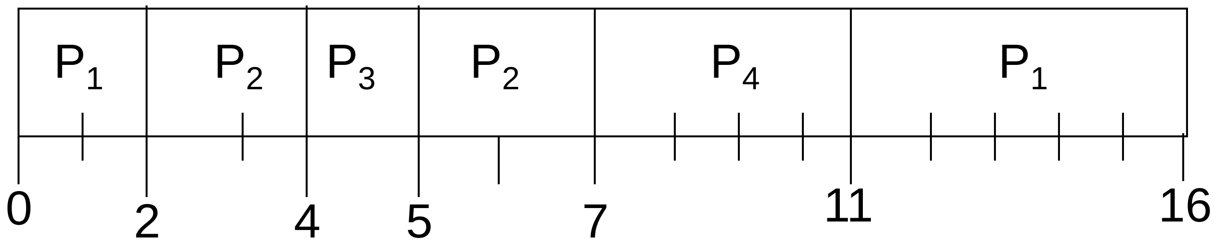
- Scheduler must have an estimate of processing time to perform the selection function.

Shortest Job First Preemptive or Shortest Remaining Time: characteristics

- **Selection Function**: minimum total service time required by the process, minus time spent in execution so far.
- **Decision Mode :** Preemptive (At arrival time)
- **Throughput:** High
- **Response Time:** Provides good response time
- **Overhead:** Can be high
- **Effect on Processes:** Penalizes long processes.
- **Starvation:** Possible

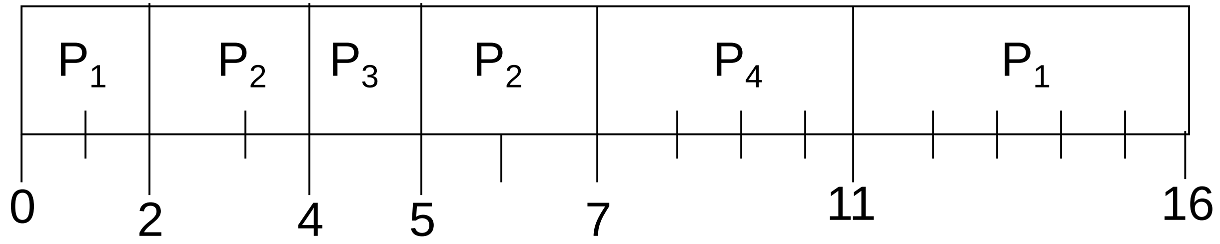
Example of Preemptive SJF

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



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**



- Average waiting time = (9 + 1 + 0 +2)/4 = 3
- Average turnaround time= (16+5+1+6)/4=7

Shortest Remaining Time Next (SRTN) (Pre-emptive)_Example

Process	Arrival Time (T0)	CPU Burst Time (in milliseconds) (Time required for completion ΔT)
P0	0	10
P1	1	6
P2	3	2
P3	5	4

Gantt Chart

P0	P1	P2	P1	P3	P0

0 1 3 5 9 13 22

- Initially only process P0 is present and it is allowed to run
- But when process P1 comes, it has shortest remaining run time.
- So, P0 is pre-empted and P1 is allowed to run.
- Whenever new process comes or current process blocks, such type of decision is taken.
- This procedure is repeated till all processes complete their execution.

Shortest Remaining Time Next (SRTN) (Pre-emptive)_Example

Output

Process	Arrival Time (T0)	Burst Time (ΔT)	Finish Time (T1)
P0	0	10	22
P1	1	6	9
P2	3	2	5
P3	5	4	13

Gantt Chart

P0	P1	P2	P1	P3	P0

0 1 3 5 9 13 22

Shortest Remaining Time Next (SRTN) (Pre-emptive)_Example

Output

Process	Arrival Time (T0)	Burst Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT = T1 - T0)
P0	0	10	22	22
P1	1	6	9	8
P2	3	2	5	2
P3	5	4	13	8

Gantt Chart

P0	P1	P2	P1	P3	P0

0 1 3 5 9 13 22

- Average Turnaround Time= (22+8+2+8) / 4 = 10 milliseconds

Shortest Remaining Time Next (SRTN) (Pre-emptive)_Example

Output

Process	Arrival Time (T0)	Burst Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT = T1 - T0)	Waiting Time (WT = TAT - ΔT)
P0	0	10	22	22	12
P1	1	6	9	8	2
P2	3	2	5	2	0
P3	5	4	13	8	4

Gantt Chart

P0	P1	P2	P1	P3	P0

0 1 3 5 9 13 22

- Average Turnaround Time= (22+8+2+8) / 4 = 10 milliseconds
- Average Waiting Time = (12+2+0+4) / 4= 4.5 milliseconds

Priority scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example

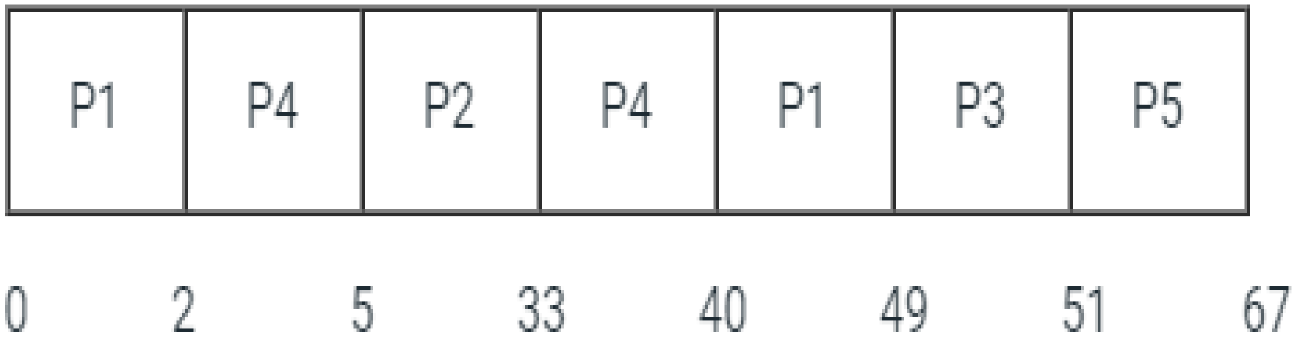
Consider the set of processes with arrival time (in milliseconds), CPU burst time (in milliseconds) , and priority (0 is the highest priority) shown below. None of the processes have I/O burst time.

Process	Arrival time	Burst Time	Priority
P1	0	11	2
P2	5	28	0
P3	12	2	3
P4	2	10	1
P5	9	16	4

The average waiting time (in milliseconds) of all the processes using preemptive priority scheduling algorithm is __________.

Process	Arrival Time	Burst Time	Priority
P_1	0	11	2
P_2	5	28	0
P_3	12	2	3
P_4	2	10	1
P_5	9	16	4

Gantt chart:



Process Table:

Process	Arrival Time	Burst Time	Priority	Completion time (CT)	Turnaround time (TAT) TAT= CT – AT	Waiting time (WT) WT = TAT-BT
P_1	0	11	2	49	49	38
P_2	5	28	0	33	28	0
P_3	12	2	3	51	39	37
P_4	2	10	1	40	38	28
P_5	9	16	4	67	58	42

Example

Consider the set of processes with arrival time (in milliseconds), CPU burst time (in milliseconds) , and priority (0 is the highest priority) shown below. None of the processes have I/O burst time.

Process	Arrival time	Burst Time	Priority
P1	0	11	2
P2	5	28	0
P3	12	2	3
P4	2	10	1
P5	9	16	4

The average waiting time (in milliseconds) of all the processes using preemptive priority scheduling algorithm is __________.

$$\text{Average waiting time} = \frac{\sum waiting\ time\ of\ all\ the\ processes}{number\ of\ processes}$$

$$= \frac{38+0+37+28+42}{5} = 29 \text{ milliseconds}$$

Answer
=29

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*)
- After time has elapsed, the process is preempted and added to the end of the ready queue.
- 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.

RR: characteristics

- **Selection Function**: constant
- **Decision Mode :** Preemptive (At time quantum)
- **Throughput:** May be low if time quantum is too small
- **Response Time:** Provides good response time for short processes
- **Overhead:** Minimum
- **Effect on Processes:** Fair treatment
- **Starvation:** No

Example of RR with Time Quantum = 20

Process	Burst Time
P1	53
P2	17
P3	68
P4	24

Example of RR with Time Quantum = 20

<u>Process</u>	<u>Burst Time</u>
P1	53
P2	17
P3	68
P4	24

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

- Average waiting time = (81+ 20 + 94+ 97)/4 = 73
- Average turnaround time= (134+37+162+121)/4=113.5

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

Example: Round Robin (By Default preemptive: Time Quantum 2)

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: Round Robin (By Default preemptive: Time Quantum 2)

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**

P1	P2	P1	P3	P2	P4	P1	P4	P1

0 2 4 6 7 9 11 13 15 16

Example: Round Robin (By Default preemptive: Time Quantum 2)

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

P1	P2	P1	P3	P2	P4	P1	P4	P1

0 2 4 6 7 9 11 13 15 16

Average waiting time = (9 + 3 + 2 +6)/4 =5
Average turnaround time =(16+7+3+10)/4=9

Consider the following four processes with arrival times (in milliseconds) and their length of CPU bursts (in milliseconds) as shown below:

Process	P1	P2	P3	P4
Arrival time	0	1	3	4
CPU burst time	3	1	3	Z

These processes are run on a single processor using preemptive Shortest Remaining Time First scheduling algorithm. If the average waiting time of the processes is 1 millisecond, then the value of Z is______.

Assume that: P4 < P3

Gantt chart:

P1	P2	P1	P1	P4	P3

0 1 2 3 4 4+Z 7+Z

Process Table1:

Process	Arrival Time (AT)	Burst Time (BT)	Completion time (CT)	Turnaround time TAT = CT - AT	Waiting time WT = TAT - BT
P1	0	3	4	4	1
P2	1	1	2	1	0
P3	3	3	7 + Z	4 + Z	1 + Z
P4	4	z	4 + Z	Z	0

Average waiting time = 1

$$\frac{1+0+(1+z)+0}{4} = 1$$

The value of Z is 2.

Assume that: P4 ≥p3

Gantt chart:

P1	P2	P1	P1	P3	P4

0 1 2 3 4 7 7+Z

Process Table2:

Process	Arrival Time (AT)	Burst Time (BT)	Completion time (CT)	Turnaround time TAT = CT - AT	Waiting time WT = TAT - BT
P1	0	3	4	4	1
P2	1	1	2	1	0
P3	3	3	7	4	1
P4	4	Z	7 + Z	Z + 3	3

Given average waiting time = 1

$But\ \frac{1+0+1+3}{4} \neq 1$

$\frac{5}{4} \neq 1$

$\therefore p4 > p3\ is\ not\ possible$

$\therefore$ **Z = 2**

Exercise

Consider the following four processes with arrival times (in milliseconds) and their length of CPU bursts (in milliseconds) as shown below:

Process	P1	P2	P3	P4
Arrival time	0	1	3	4
CPU burst time	3	1	3	Z

These processes are run on a single processor using preemptive Shortest Remaining Time First scheduling algorithm. If the average waiting time of the processes is 1 millisecond, then the value of Z is______.

Answer
Z=2

Exercise

Considrer the following CPU processes with arrival times (in miliseconds) and length of CPU bursts (in miliseconds) as given below:

Process	Arrival time	Burst time
P1	0	7
P2	3	3
P3	5	5
P4	6	2

If the pre-emptive shortest remaining time first scheduling algorithm is used to schedule the processes, then the average waiting time across all processes is ______ milliseconds.

Considrer the following CPU processes with arrival times (in miliseconds) and length of CPU bursts (in miliseconds) as given below:

Process	Arrival time	Burst time
P1	0	7
P2	3	3
P3	5	5
P4	6	2

If the pre-emptive shortest remaining time first scheduling algorithm is used to schedule the processes, then the average waiting time across all processes is ______ milliseconds.

Answer
=3

Exercise

Consider the following processes, with the arrival time and the length of the CPU burst given in milliseconds. The scheduling algorithm used is preemptive shortest remaining-time first.

Process	Arrival Time	Burst Time
P_1	0	10
P_2	3	6
P_3	7	1
P_4	8	3

The average turn around time of these processes is milliseconds _______.

Exercise

Consider the following processes, with the arrival time and the length of the CPU burst given in milliseconds. The scheduling algorithm used is preemptive shortest remaining-time first.

Process	Arrival Time	Burst Time
P_1	0	10
P_2	3	6
P_3	7	1
P_4	8	3

The average turn around time of these processes is milliseconds ______.

Answer
=8.25

Exercise

For the processes listed in the following table, which of the following scheduling schemes will give the lowest average turnaround time?

Process	Arrival Time	Processing Time
A	0	3
B	1	6
C	4	4
D	6	2

(A) First Come First Serve

(B) Non – preemptive Shortest Job First

(C) Shortest Remaining Time

(D) Round Robin with Quantum value two

Exercise

For the processes listed in the following table, which of the following scheduling schemes will give the lowest average turnaround time?

Process	Arrival Time	Processing Time
A	0	3
B	1	6
C	4	4
D	6	2

(A) First Come First Serve (B) Non – preemptive Shortest Job First (C) Shortest Remaining Time

(D) Round Robin with Quantum value two

Answer (C) Shortest Remaining Time

Exercise

Consider the 3 processes, P1, P2 and P3 shown in the table.

Process	Arrival time	Time Units Required
P1	0	5
P2	1	7
P3	3	4

The completion order of the 3 processes under the policies FCFS and RR2 (round robin scheduling with CPU quantum of 2 time units) are

- (A) **FCFS:** P1, P2, P3 **RR2:** P1, P2, P3
- (B) **FCFS:** P1, P3, P2 **RR2:** P1, P3, P2
- (C) **FCFS:** P1, P2, P3 **RR2:** P1, P3, P2
- (D) **FCFS:** P1, P3, P2 **RR2:** P1, P2, P3

MIT-WPU

॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Exercise

Consider the 3 processes, P1, P2 and P3 shown in the table.

Process	Arrival time	Time Units Required
P1	0	5
P2	1	7
P3	3	4

The completion order of the 3 processes under the policies FCFS and RR2 (round robin scheduling with CPU quantum of 2 time units) are

- (A) **FCFS:** P1, P2, P3 **RR2:** P1, P2, P3
- (B) **FCFS:** P1, P3, P2 **RR2:** P1, P3, P2
- (C) **FCFS:** P1, P2, P3 **RR2:** P1, P3, P2
- (D) **FCFS:** P1, P3, P2 **RR2:** P1, P2, P3

Answer (C) **FCFS:** P1, P2, P3 **RR2:** P1, P3, P2

In RR, time slot is of 2 units. Processes are assigned in following order

p1, p2, p1, p3, p2, p1, p3, p2, p2

This question involves the concept of ready queue. At t=2, p2 starts and p1 is sent to the ready queue and at t=3 p3 arrives so then the job p3 is queued in ready queue after p1.

So at t=4, again p1 is executed then p3 is executed for first time at t=6

References

1. William Stallings, Operating System: Internals and Design Principles, Prentice Hall, ISBN-10: 0-13-380591-3, ISBN-13: 978-0-13-380591-8, 8th Edition

2. Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, WILEY,ISBN 978-1-118-06333-0, 9th Edition

Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

Process Management

Syllabus OS [2018-2019]--Trimester VI

Process Management

Process: Concept of a Process, Process States, Process Control-creation, new program execution, termination.

Threads: Processes and Threads, Concept of Multithreading, Types of Threads, Thread programming Using Pthreads.

Scheduling: Types of Scheduling, Scheduling Algorithms: FCFS, SJF, Priority, Round Robin.

References

1. William Stallings, Operating System: Internals and Design Principles, Prentice Hall, ISBN-10: 0-13-380591-3, ISBN-13: 978-0-13-380591-8, 8th Edition

2. Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, WILEY,ISBN 978-1-118-06333-0, 9th Edition

Process Management

- **Concept of a Process :** Process is an instance of a program in execution
- It is an entity that can be assigned to and executed on a processor
- **A process is comprised of:**
  - Program code/instructions
  - Data
  - Stack
  - A number of attributes describing the state of the process
- When a process is mapped on to the memory it has an address space. This address space includes the code , data and stack for the process
- Terms *job, task* and *process* are used almost interchangeably.
- Many copies of editor program(passive entity) invoked, each is a separate process(active entity)

Process Management

What is process management ?

- The processes are represented and controlled by the OS, this is known as process management.
- **The Process states** which characterize the behaviour of processes.
- **The Data structures** that are used to manage processes.
- It describes the ways in which the OS uses these data structures to control process execution.

Process Management tasks of an Operating System

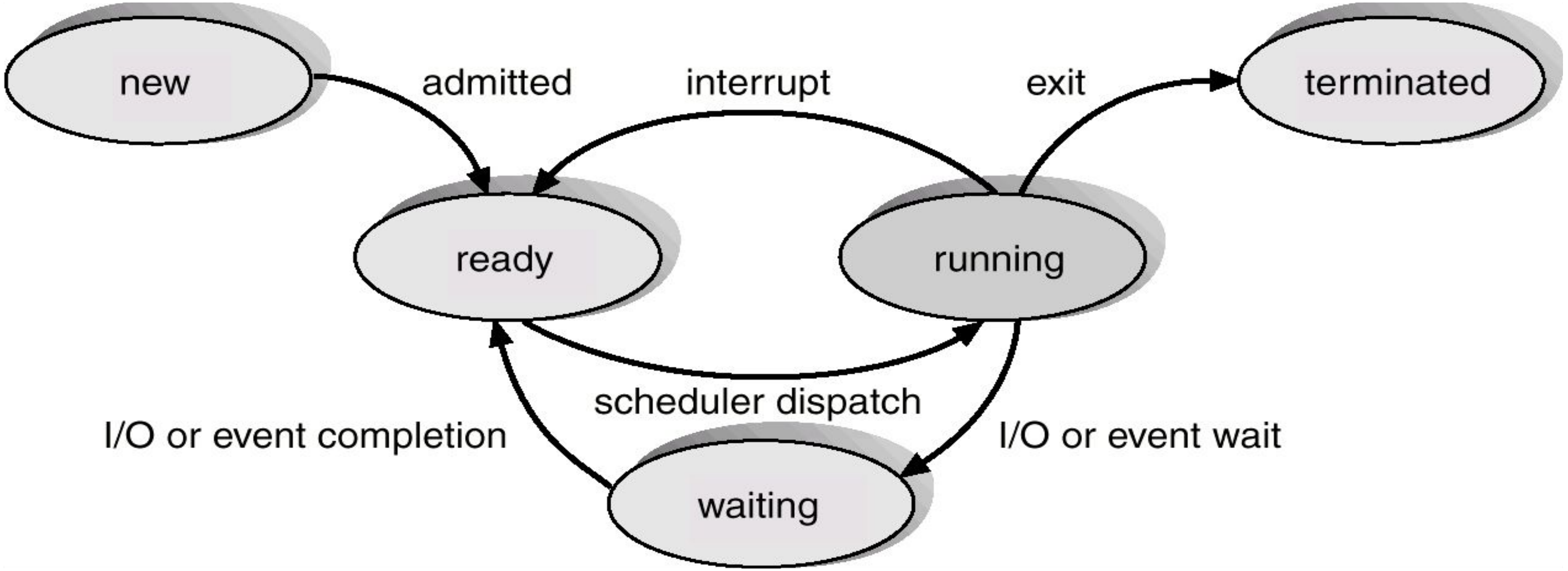
- Interleave the execution of multiple processes
- Allocate resources to processes, and protect the resources of each process from other processes,
- Enable processes to share and exchange information,
- Enable synchronization among processes.

Process States

When a program executes as a process it goes through multiple states before it completes execution

- **new:** process is created
- **ready:** process is waiting to be assigned processor. The process has every other resource except the processor
- **running:** Instructions are being executed. State has all resources including the processor
- **waiting:** process is waiting for some event to occur (eg. I/O completion). When an executing process needs an I/O device/services, it gets into wait state and when the i/o requirement is fulfilled it goes back into ready state
- **terminated:** process has finished execution

Diagram for Process States



Suspended State

- Processor is faster than I/O so many processes could be waiting for I/O
- Swap these processes to disk to free up more memory
- Ready/Waiting state becomes ***suspend*** state when swapped to disk

Process Control Block (PCB)

Process Control Block [PCB] : It is a Data-structure maintained by the Operating System. It holds all necessary information related to a Process.

Information associated with each process is a follows:-

Process state

Program counter

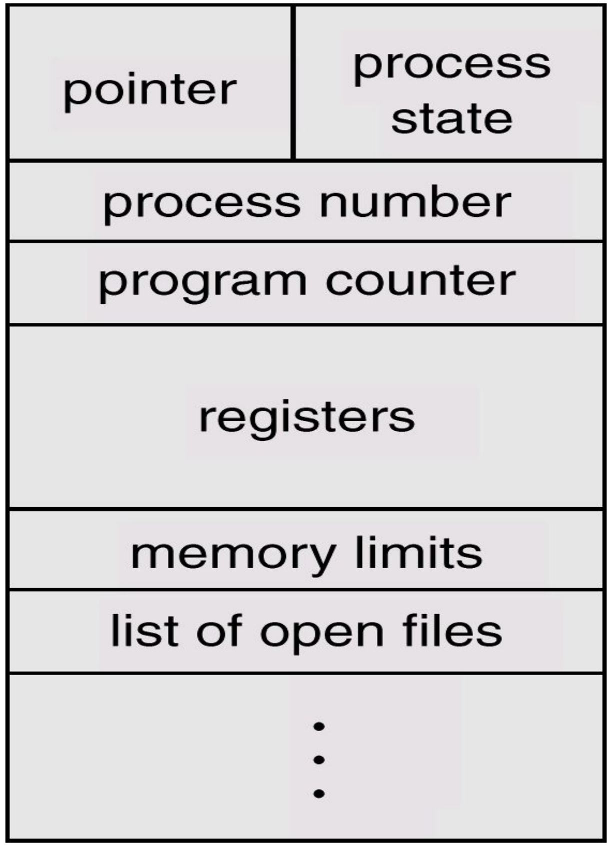
CPU registers

CPU scheduling information

Memory-management information

Accounting information

I/O status information



Process Modes

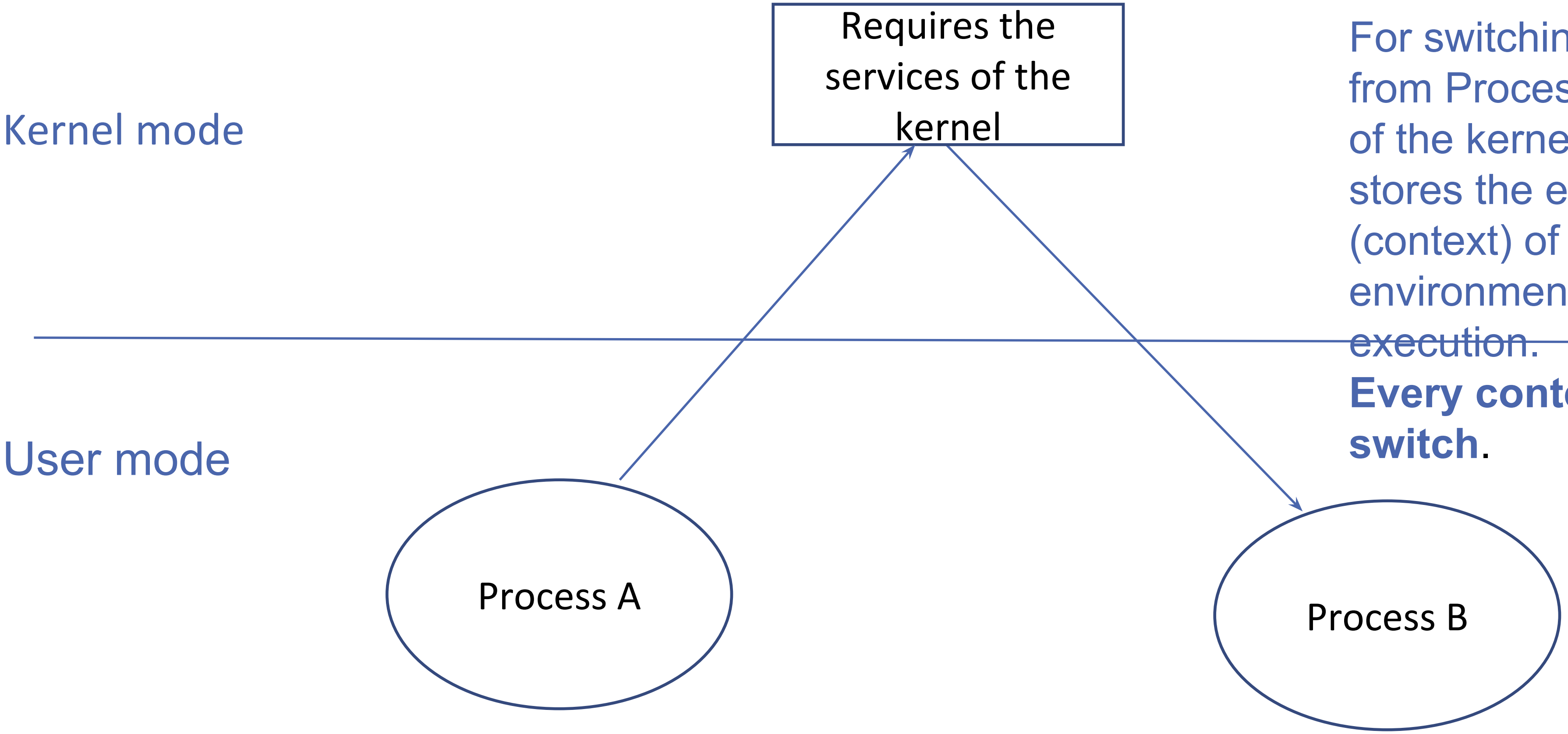
There are two modes of operations :-

Kernel mode (Privileged mode)..it can access its own data-structures as well as the user mode data structures.

User mode : it can access only the user mode data structures.

- User programs initially work in the User mode.
- Whenever a system call is encountered the control switches to the kernel mode.
- All interrupts are serviced in the Kernel mode.
- When the system call is serviced the control returns back to the user mode

Process Management --- context switch



For switching of context (environment) from Process A to process B, services of the kernel are needed. The kernel stores the execution environment (context) of process A and retrieves the environment(context) for process B for execution.

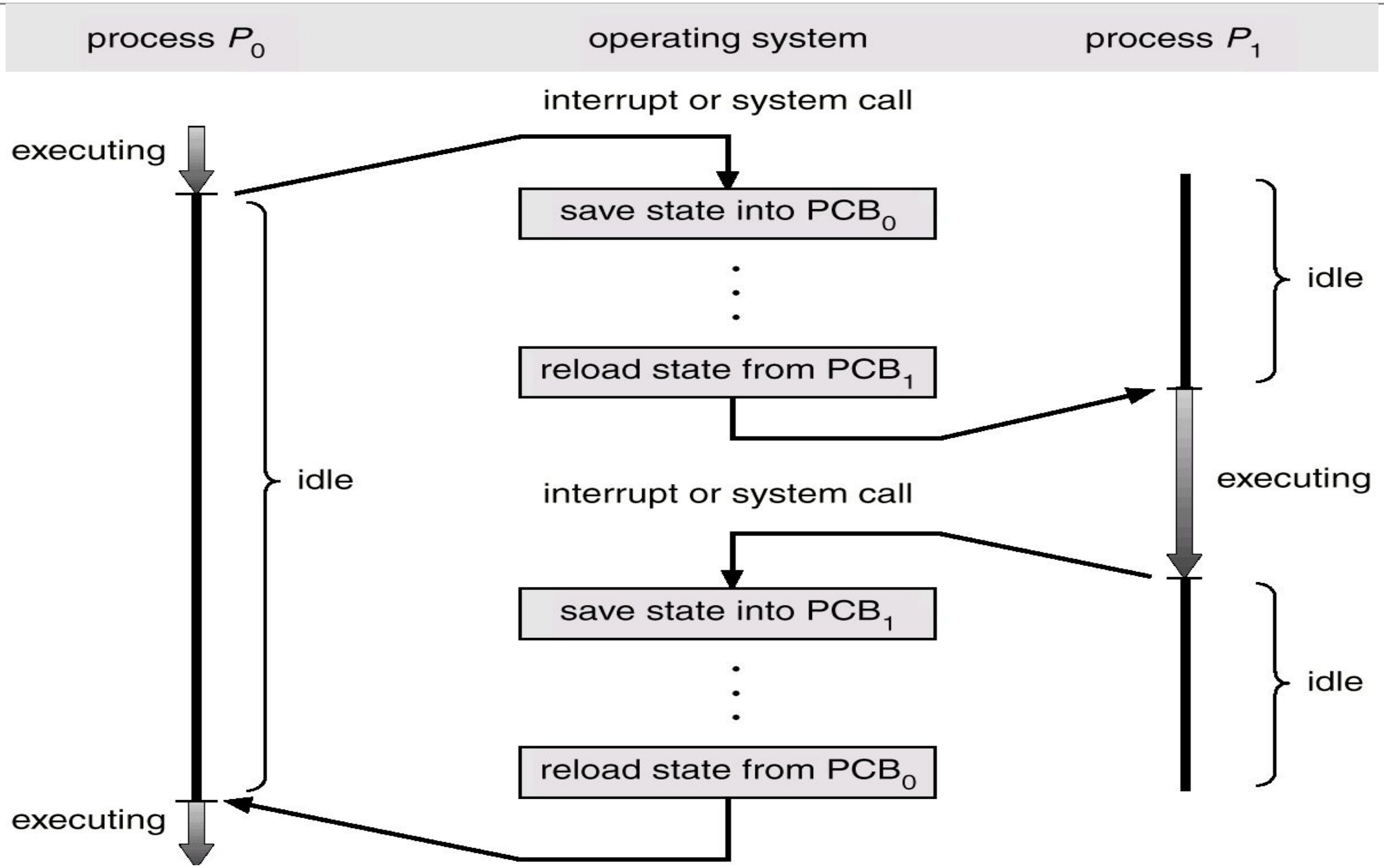
Every context switch is a mode switch.

Context Switch

When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.

Context-switch time is overhead; the system does no useful work while switching.

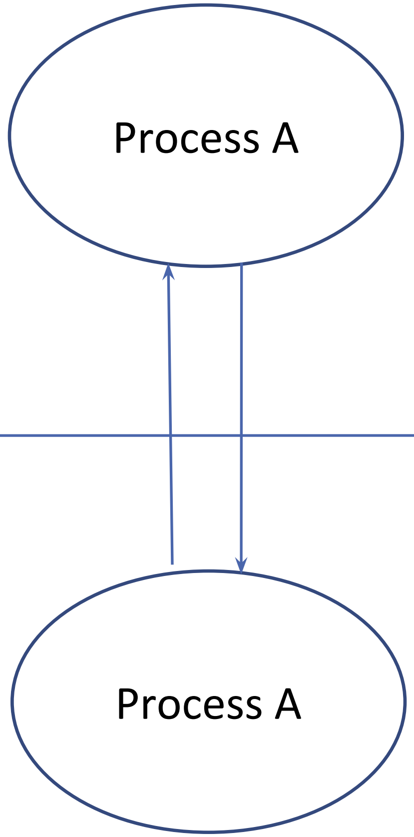
Context Switch



Process Management – Mode switch

Kernel mode

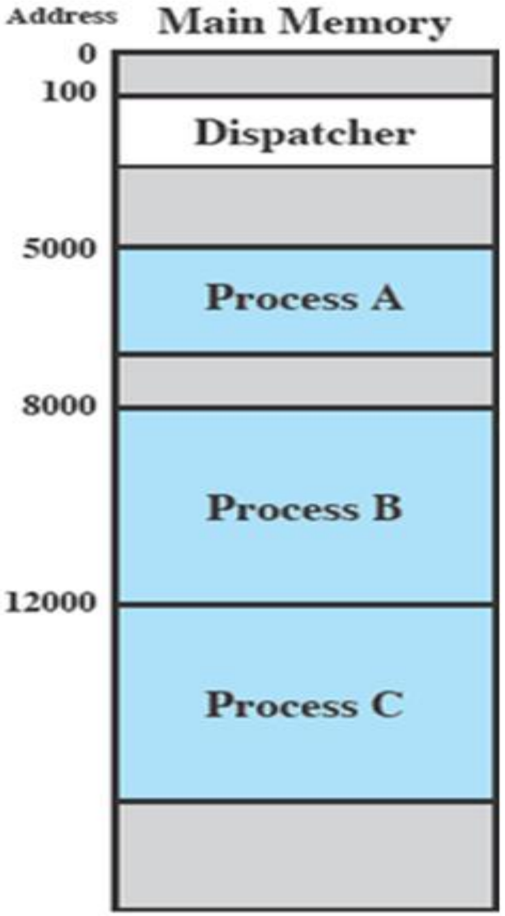
User mode



Process A was executing in User mode and has now switched to the Kernel mode due to some system call.This is mode switch.

As, every context switch is a mode switch, every mode switch may/may not be a context switch.

Process Execution



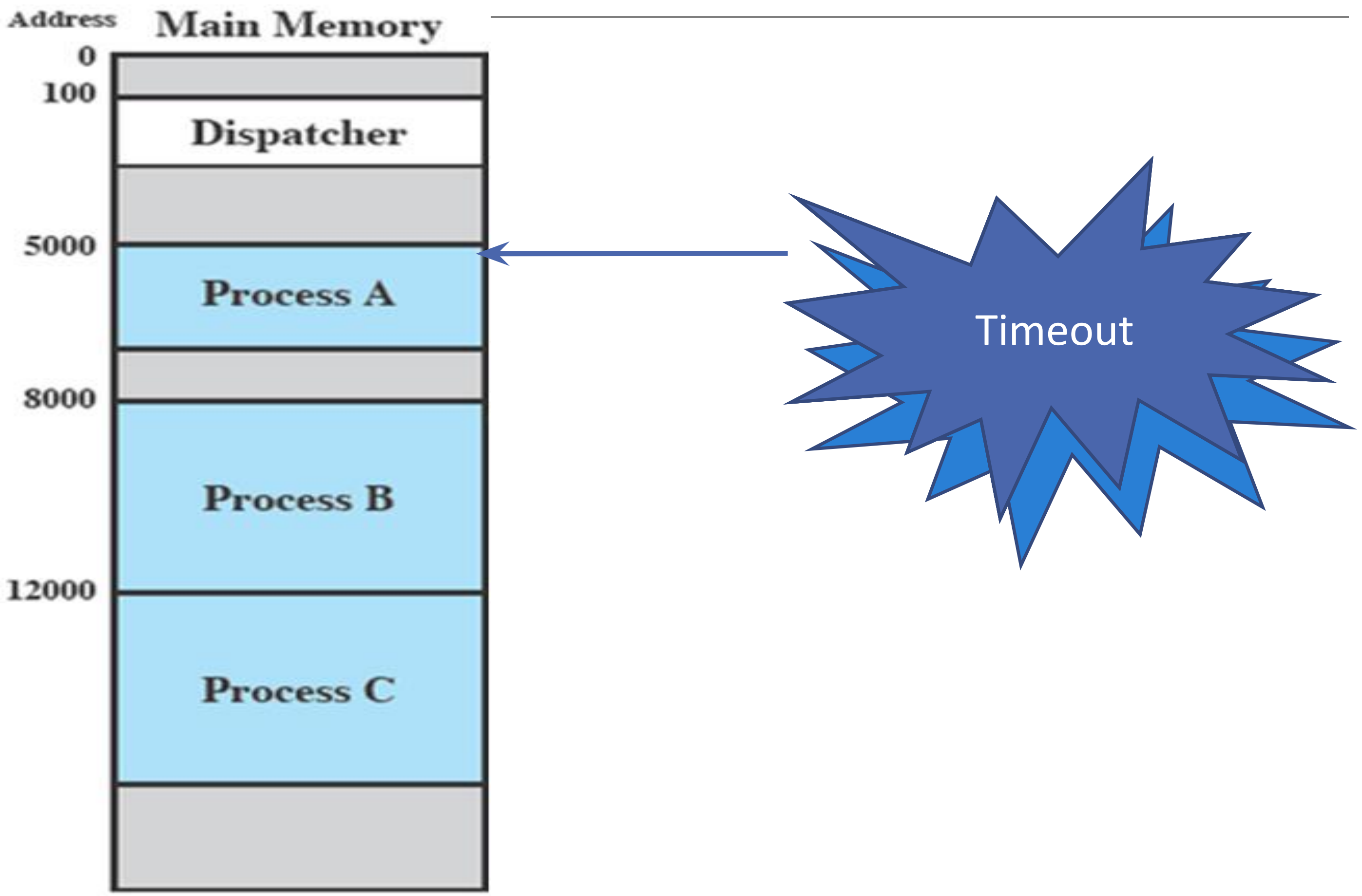
Consider three processes being executed

All are in memory (plus the dispatcher)

Dispatcher is a small program which switches the processor from one process to another

Selecting a process among various processes is done by **scheduler**. Here the task of scheduler completed. Now **dispatcher** comes into picture as scheduler have decide a process for execution, it is dispatcher who takes that process from ready queue to the running status, or providing CPU to that process is the task of dispatcher.

Process Execution



1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
----- Timeout	
7	100
8	101
9	102
10	103
11	104
12	105
13	8000
14	8001
15	8002
16	8003
----- I/O Request	
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003

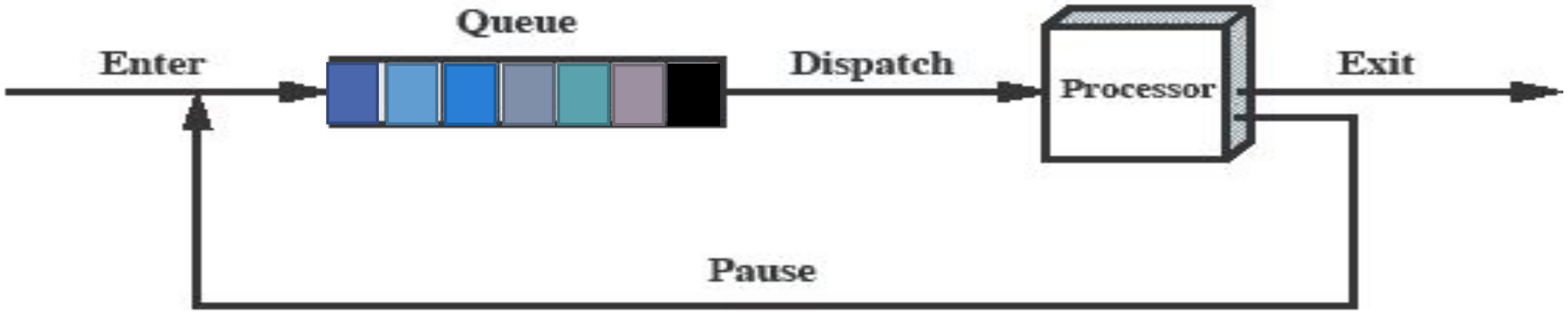
27	12004
28	12005
----- Timeout	
29	100
30	101
31	102
32	103
33	104
34	105
35	5006
36	5007
37	5008
38	5009
39	5010
40	5011
----- Timeout	
41	100
42	101
43	102
44	103
45	104
46	105
47	12006
48	12007
49	12008
50	12009
51	12010
52	12011
----- Timeout	

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

Queuing Diagram



(b) Queuing diagram

Processes moved by the dispatcher of the OS to the CPU then back to the queue until the task is competed

Process Creation

When a new process is created , the following happens :-

- Allocates space to the process in memory.
- Assign a unique process ID to the process
- A Process control block (PCB) gets associated with the process.
- OS maintains pointers to each process's PCB in a process table so that it can access the PCB quickly.

Reasons to create a new process

- New user Job
- Created by O/S to provide a service
- Spawned by existing process: The action of creating a new process (Child Process) at the explicit request of another process (Parent Process) is called as process spawning. E.g A print server or file server may generate a new process for each request that it handles

After Creation

After creating the process the Kernel can do one of the following:

- Stay in the parent process.
- Transfer control to the child process
- Transfer control to another process.

Process Creation

Fork

System call **fork()** is used to create processes. It takes no arguments and returns a process ID.

The syntax for the fork system call

- pid = fork();
  - In the parent process, pid is the child process ID
  - In the child process, pid is 0

Sequence of operations for fork.

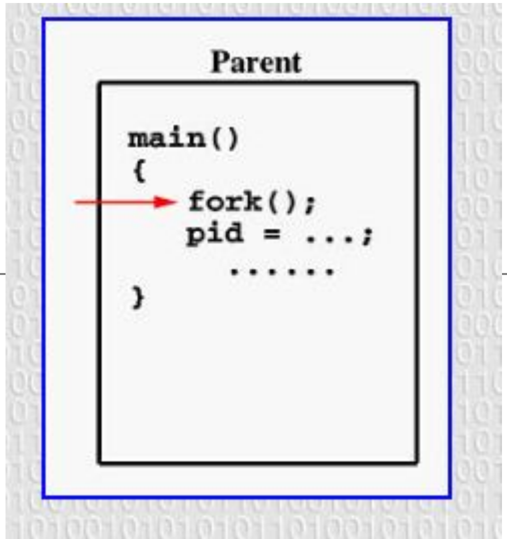
- 1. It allocates a slot in the process table for the new process
- 2. It assigns a unique ID number to the child process
- 3. It makes a copy of the context of the parent process.
- 4. It returns the ID number of the child to the parent process, and a 0 value to the child process.

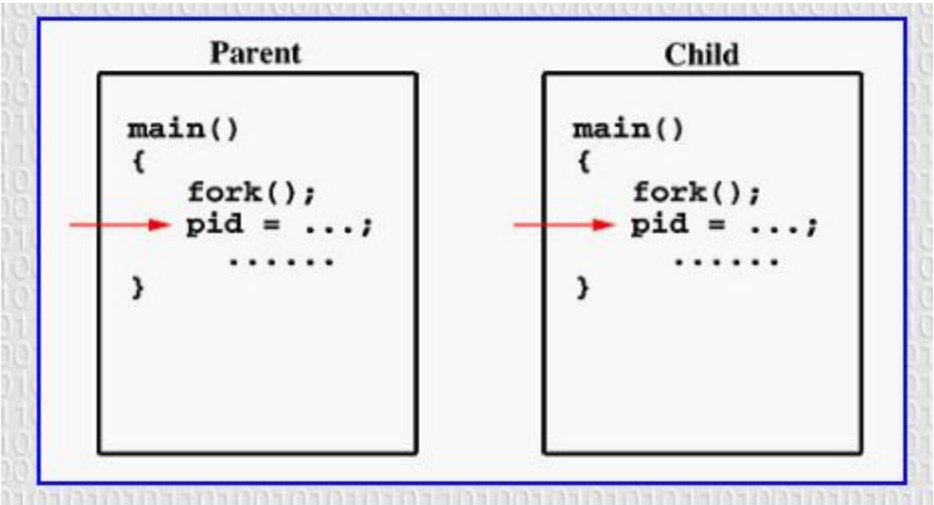
Fork

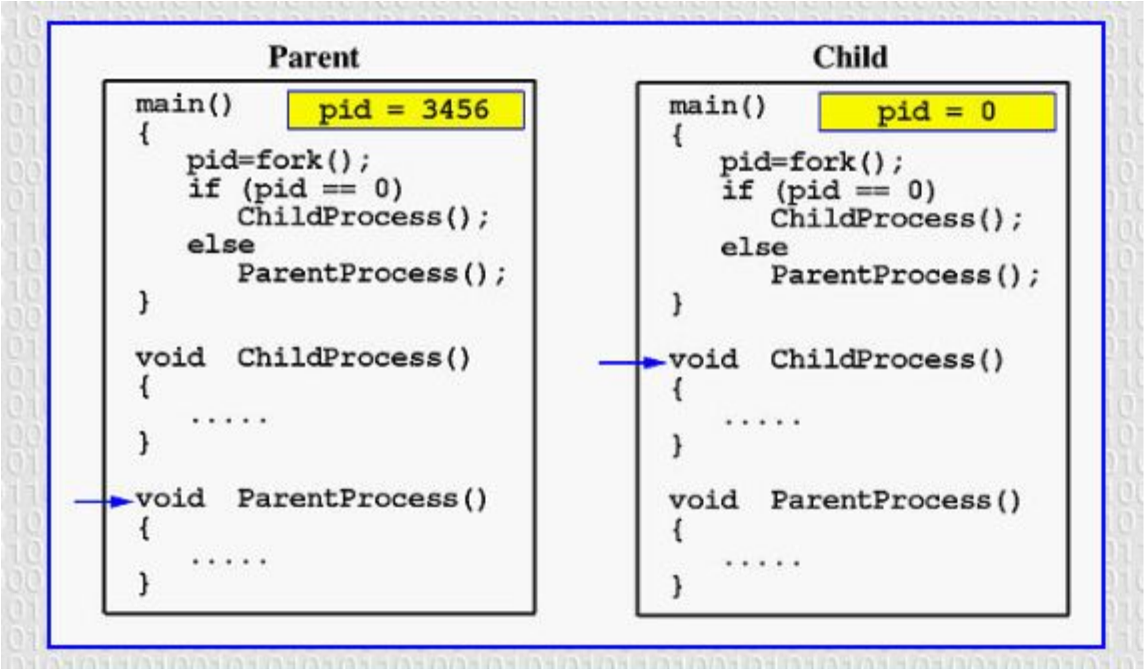
- Purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller.
- After a process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call.
- To distinguish the parent from the child, the returned value of **fork()** can be used:

- **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent

- Returned process ID is of type **pid_t** defined in **sys/types.h**
- Process can use function **getpid()** to retrieve the process ID assigned to this process
- **Unix/Linux will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces**.


Parent
main()
{
fork();
pid = ...;
......
}


Parent
main()
{
fork();
pid = ...;
......
}
Child
main()
{
fork();
pid = ...;
......
}


Parent
main()
pid = 3456
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}
Child
main()
pid = 0
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}

Process Management—creating a Process in Unix(example)

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

/* pid_t : is a long integer type data type...prototype in types.h */
pid_t num_pid;

main()
{
num_pid=fork(); /* return value of fork */

if(num_pid==0) /* this is child process */
{
 printf("this is the child process id %d\n",getpid());

}

if(num_pid>0) /* this is parent process */
{
printf("this is the parent process id %d",getpid());

}
exit();
}
```

```
$cc program.c
$ ./a.out
$ this is the child process id 1001
$ this is the parent process id 1000
```

Process Creation – Parent/Child

Parent Process	Child Process
The parent process has its unique ID	The child process has its unique ID
The parent process creates a child process by giving a call to fork() system call	The child process gets created due to the fork() system call

The child is initially a duplication of the parent process.
The child and parent do exist in separate address spaces.
The child inherits all data structures

A client-server application can be built using the parent-child concept.
Any IPC mechanism can be implemented using the parent child relationship.

Process Termination

When a process terminates :

- All the resources held by process are released

- All the information held in all data structures is removed

- A process goes back to becoming a program and is stored on the secondary memory.

Process Management-- Threads

- A thread is a part of a program.
- It is an execution unit of the CPU.
- All threads of the same process share the same address space.
- All Threads have separate stacks and individual Thread IDs
- Thread is a lightweight process because :The context switching between threads is inexpensive in terms of memory and resources.

Process Management-- Threads

Multithreading: Ability of an OS to support multiple, concurrent paths of execution within a single process.

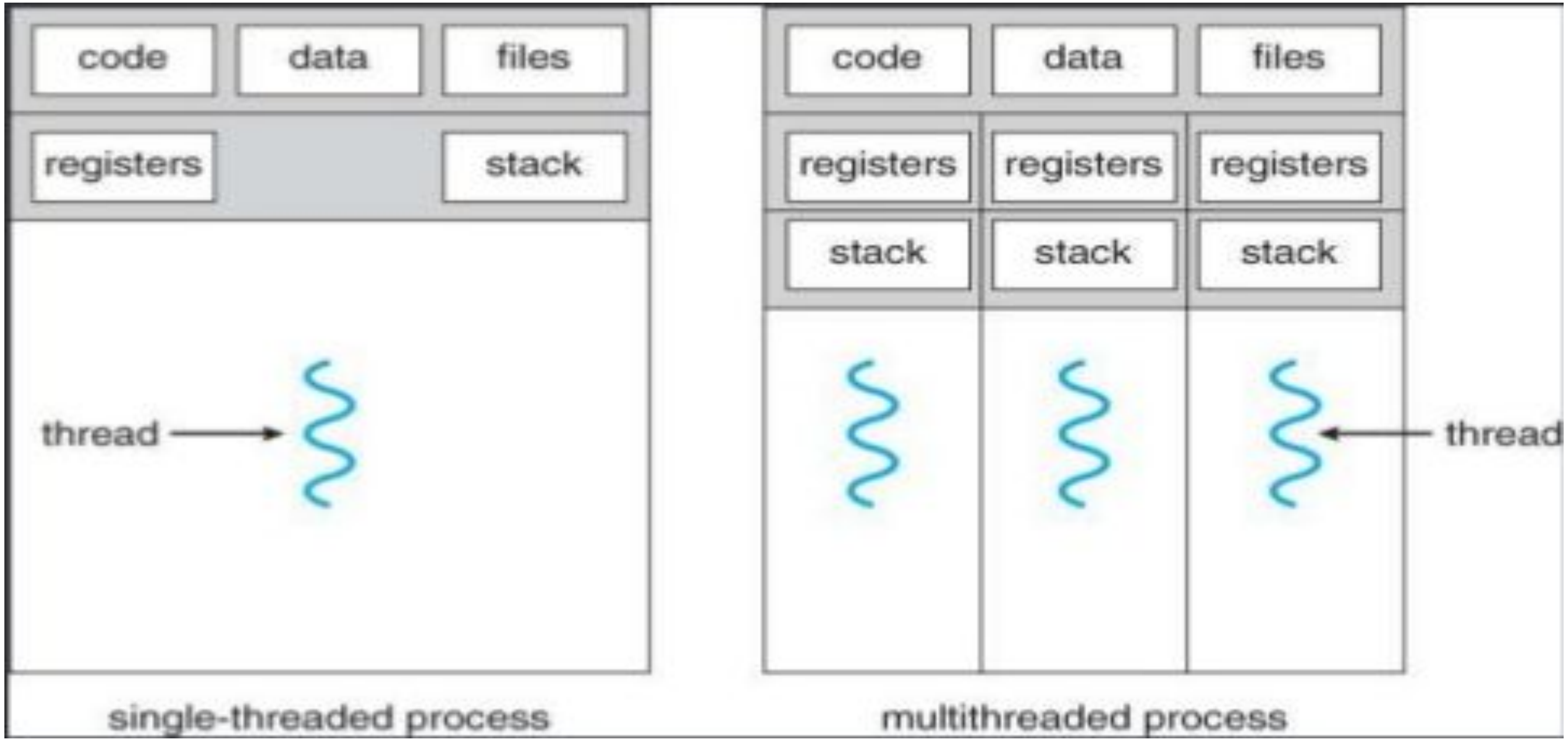
It is also described as the interleaved execution of threads.

Process Management– Differences between threads and processes

Process	Threads
A process is a program in execution	A thread is a part of the process
A process has its own Process ID	A thread has its own thread ID
Every process has its own memory space	Threads use the memory of the process they belong to
Inter process communication is slow as processes have different memory address	Inter thread communication for threads within the same process is fast
The context switching is more expensive in terms of memory and resources.	The context switching is less expensive in terms of memory and resources . Majorly because threads of the same process share the same memory space.

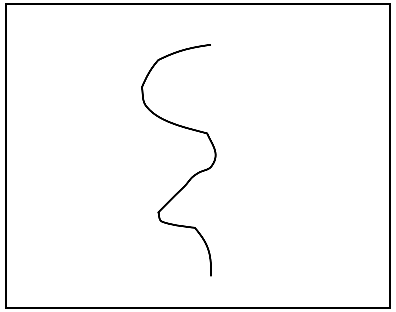
Process Management- ---

Threads, a diagrammatic representation

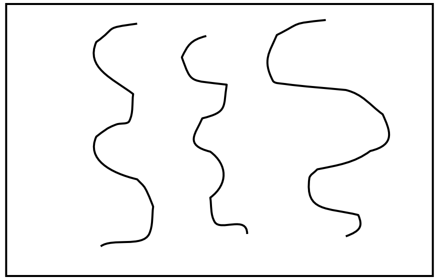


Process Management

Threads and processes diagrammatic representation

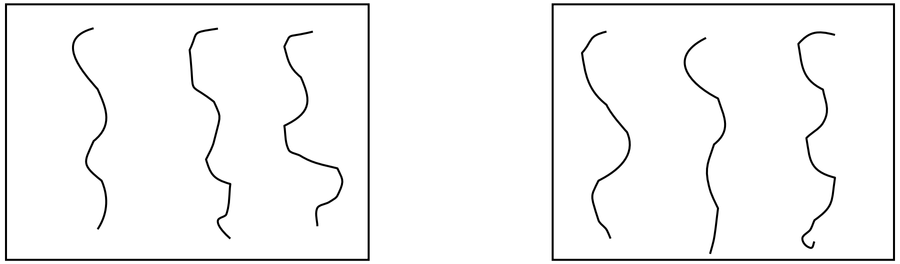


One process
one thread



One Process
Multiple threads

Multiple Processes
one thread per process



Multiple Processes
Multiple thread per process

Multithreading

- Operating system supports multiple threads of execution within a single process
- Examples:
  - MS-DOS supports a single thread
  - UNIX supports multiple user processes but only supports one thread per process
  - Java run time environment supports one process with multiple threads
  - Windows, Solaris, Linux, Mach, and OS/2 support multiple threads

Process Management—Thread basics

Thread operations include thread creation, termination, synchronization (join, blocking), scheduling, etc.

All threads within a process share the same address space.

Threads in the same process share: Process instructions
- Data
- open files (descriptors)
- signals
- current working directory
- User and group id

Each thread has a unique: Thread ID
- set of registers
- stack for local variables, return addresses
- priority

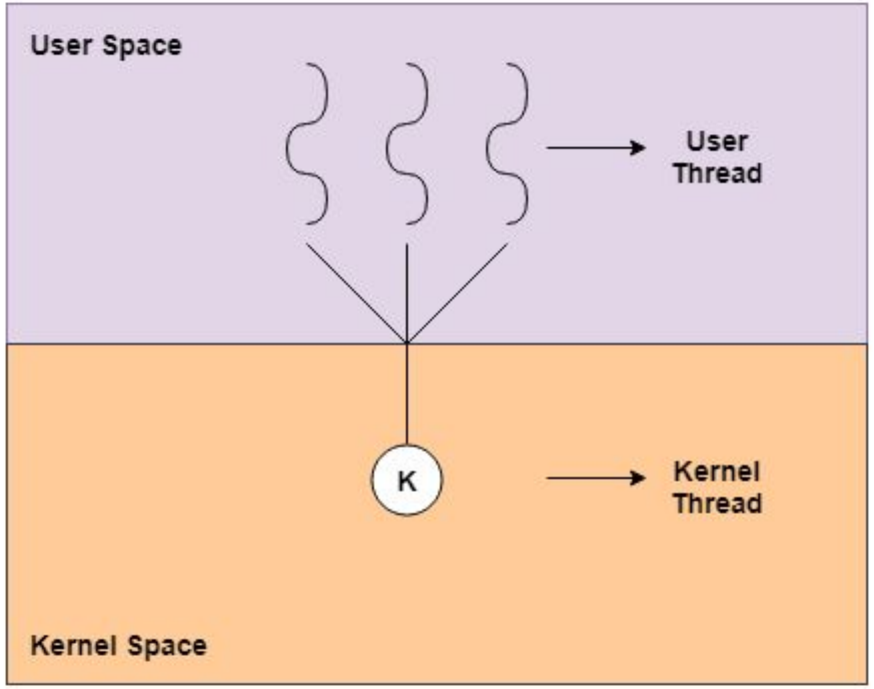
Process Management—Thread basics

Types of Threads :-

There are majorly two types of threads :-

1. kernel level threads

2. User level threads



Process Management—Thread basics

User - Level Threads

- User-level threads are implemented by users and the kernel is not aware of the existence of these threads
- Kernel handles them as if they were single-threaded processes.
- User-level threads are small and much faster than kernel level threads.
- They are represented by a program counter(PC), stack, registers and a small process control block.
- Also, there is no kernel involvement in synchronization for user-level threads.
- User-Level threads are managed entirely by the user-level library

Process Management—Thread basics

Advantages of User-Level Threads

- User-level threads are easier and faster to create than kernel-level threads. They can also be more easily managed.
- User-level threads can be run on any operating system.
- There are no kernel mode privileges required for thread switching in user-level threads.

Disadvantages of User-Level Threads

- Multithreaded applications in user-level threads cannot use multiprocessing to their advantage.
- Entire process is blocked if one user-level thread performs blocking operation.

Process Management—Thread basics

Kernel-Level Threads

- Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel

- Context information for the process as well as the process threads is all managed by the kernel.

- Because of this, kernel-level threads are slower than user-level threads.

Process Management—Thread basics

.Advantages of Kernel-Level Threads

- Multiple threads of the same process can be scheduled on different processors in kernel-level threads.
- The kernel routines can also be multithreaded.
- If a kernel-level thread is blocked, another thread of the same process can be scheduled by the kernel.

Disadvantages of Kernel-Level Threads

- A mode switch to kernel mode is required to transfer control from one thread to another in a process.
- Kernel-level threads are slower to create as well as manage as compared to user-level threads.

Process Management—POSIX pthread

Portable Operating System Interface Standard (POSIX)

```
#include<pthread.h>
int pthread_create (pthread_t *tid, const pthread_attr_t *attr, void *(*func)(void*),void *arg);
```

0 : OK +ve : error

pthread_t *tid : Returns the thread ID which is of type pthread_t, i.e. long int.

const pthread_attr_t *attr : Thread attribute list

void *(*func)(void*) : A function that works as a thread.

void *arg : A list of arguments sent to the function

Process Management—POSIX pthread
Portable Operating System Interface Standard (POSIX)

```
#include <pthread.h>
```

void pthread_exit(void **retval*);

The pthread_exit(); function terminates the calling thread and returns a value via retval that (if the thread is joinable) is available to another thread in the same process that calls pthread_join().

Process Management—POSIX pthread Portable Operating System Interface Standard (POSIX)

#include<pthread.h>

int pthread_join(pthread_t tid, void **status);

0:OK		+ve:error

pthread_join() function shall suspend execution of the calling thread until the target thread terminates

On return from a successful pthread_join() call with a non-NULL status argument, the value passed to pthread_exit() by the terminating thread shall be made available in the location referenced by status.

Process Management—POSIX pthread
Portable Operating System Interface Standard (POSIX)

#include<pthread.h>

pthread_t pthread_self(void);

Returns: thread ID of calling thread

The pthread_self() function returns the ID of the calling thread. This is the same value that is returned in *tid in the pthread_create() call that created this thread.

Process Management--POSIX pthread

#include<pthread.h>

int pthread_detach(pthread_t tid);

Returns : 0:OK   +ve: error

The **pthread_detach**() function marks the thread identified by *thread* as detached.

When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

Need for linking –lpthread flag at the time of compilation :-

-lpthread in essence tells the GCC compiler that it must link the pthread library to the compiled executable. pthread or POSIX Threads is a standardized library for implementing threads in C

pthread example

```
#include<stdio.h>
#include<pthread.h>


int add1(int a[3])

{
      a[2] = a[1] + a[0];

          printf("result from thread 1 is %d\n",a[2]);
}
int add2(int b[3])

{
      b[2] = b[1] + b[0];

       printf("result from thread 2 is %d\n",b[2]);
}
```

```
main()
{     int arr[4],a[3],b[3],i,ans=0;
      pthread_t thread1,thread2;
      printf("enter 4  numbers\n");
      for(i=0;i<4;i++)
              scanf("%d",&arr[i]);
      a[0]=arr[0];     a[1]=arr[1];

              b[0]=arr[2];     b[1]=arr[3];

      pthread_create(&thread1,NULL,(void*)add1,a);
      pthread_create(&thread2,NULL,(void*)add2,b);
      pthread_join(thread1,NULL);
      pthread_join(thread2,NULL);

      ans=a[2]+b[2];
      printf("the result = %d\n",ans);
}
```

MIT-WPU

॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

Operating Systems

School of Computer Engineering and technology

Inter-process communication (IPC)

There are several mechanisms for *Inter-Process Communication* (IPC)

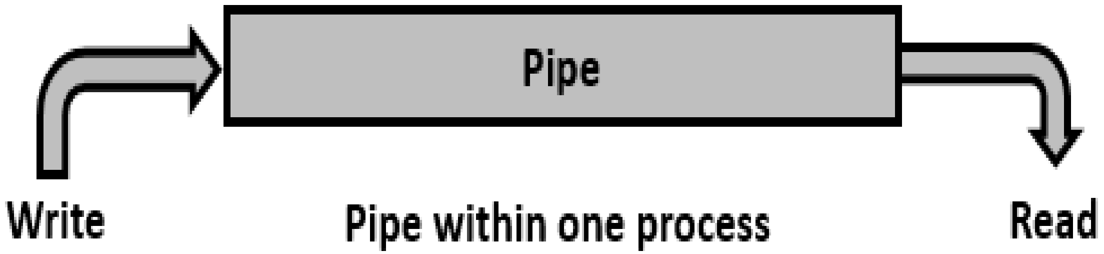
- Signals
- FIFOS (named pipes)
- Pipes
- Sockets
- Message passing
- Shared memory
- Semaphores

Pipe

Pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes.

UNIX deals with pipes the same way it deals with files.

A process can send data **'down' a pipe using a write system call** and another process can receive the **data by using read at the other end.**



Pipe within one process

Programming with pipes

- Within programs a pipe is created using a system call named **pipe.**
- This system call would create a pipe for one-way communication.
- This call would return zero on success and -1 in case of failure.
- If successful, this call returns two files descriptors:

Usage

```
#include <unistd.h>
int pipe(int filedes[2]);
```

Programming with pipes

- Usage

```
#include <unistd.h>
int pipe(int filedes[2]);
```

- filesdes is a two-integer array that will hold the file descriptors that will identify the pipe If successful,
- filedes[0] will be open for reading from the pipe and
- filedes[1] will be open for writing down it.
- pipe can fail (returns -1) if it cannot obtain the file descriptors (exceeds user-limit or kernel-limit).

Programming with pipes

Here's an extremely important point: a read from a pipe only gives end-of-file if *all* file descriptors for the write end of the pipe have been closed.

Thus, after a fork, whichever process is intending to do the reading (and thus not the writing) had best close the write end of the pipe.

```
#include<unistd.h>
close(filedes)
```

The above system call closing already opened file descriptor. This implies the file is no longer in use and resources associated can be reused by any other process.

Programming with pipes

write(filedes[1], string, MAX);

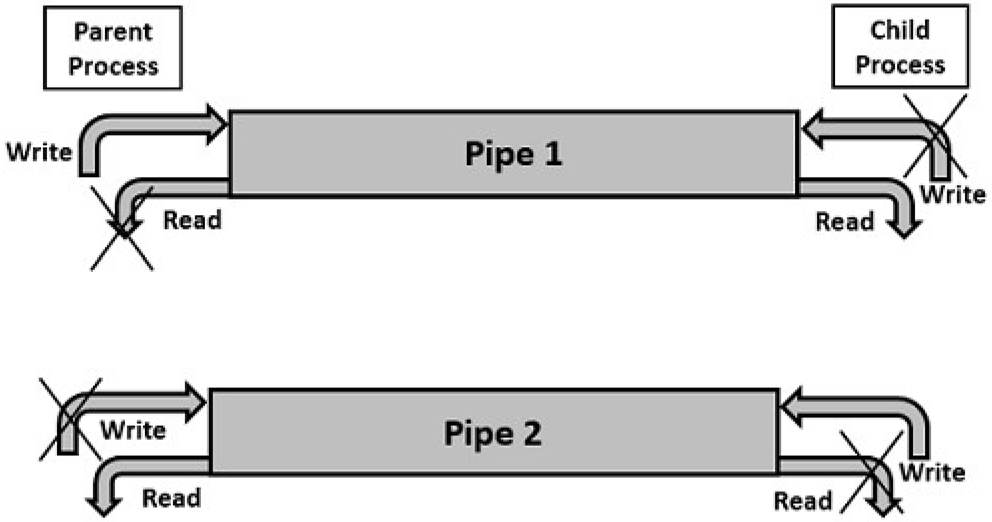
- The above system call is to write to the specified file with arguments of the file descriptor fd, string and the size of buffer.
- The file descriptor id is to identify the respective file, which is returned after calling pipe() system call.
- The file needs to be opened before writing to the file. It automatically opens in case of calling pipe() system call.
- This call would return the number of bytes written (or zero in case nothing is written) on success and -1 in case of failure. Proper error number is set in case of failure.

Programming with pipes

read(filedes[0], line, MAX);

- The above system call is to read from the specified file with arguments of file descriptor fd, string and the size of buffer.

Two-way Communication Using Pipes



- Pipe communication is viewed as only one-way communication i.e., either the parent process writes and the child process reads or vice-versa but not both.
- However, what if both the parent and the child needs to write and read from the pipes simultaneously, the solution is a two-way communication using pipes. Two pipes are required to establish two-way communication.