Page No:
Date:

What is Operating System ?

① Intermediatory - Acts as the intermediatory b/w user and hardware

② Resource manager / allocator - Operating system controls and coordinates the use of system resources among various application programs in an unbiased fashion

③ Platform - OS provides the platforms on which other application programs can be installed, provides the environment within which programs are executed

- Application software performs specific tasks for the user
- System software operates and controls computer system and provides a platform to run application software

Operating system functions -

(i) Access to computer hardware
(ii) Interface b/w user and computer hardware
(iii) Resource management (AKa, arbitration) (memory, device, file, security, process etc)
(iv) Hides the underlying complexity of the hardware (AKa, abstraction)
(v) Facilitates execution of application programs by providing isolation and protection.

OS goals -

- Maximum CPU utilization
- less process starvation
- Higher priority job execution

Major components of operating systems

① Kernel
- Central component : Manages the system's resources and communication b/w hardware and software.

② Process management
- Process scheduler : Determines execution of process

Page No:
Date:

- Process control block (PCB): contains process details such as process ID, priority, status, etc.
- Concurrency control: Manages simultaneous execution.

③ Memory management
- Physical memory management: - Manages RAM allocation
- Virtual memory management - Stimulates additional memory using disk space.
- Memory allocation - Assigns memory to different processes

④ File system management
- File handling - Manages the creation, deletion and access of files and directories.
- File control block - Stores file attributes and control information
- Disk scheduling - Organises the order of reading or writing to disk

⑤ Device management
- Device drivers: Interface b/w hardware and the OS
- I/O Controllers: Manage data transfer to and from peripheral devices.

⑥ Security and Access control
- Authentication: Verifies user credentials
- Authorization: Controls access permissions to files / directories
- Encryption: Ensures data confidentiality and integrity

⑦ User interface
- Command line Interface (CLI): text based user interaction
- Graphical User Interface (GUI): Visual, user friendly interaction with the OS.

⑧ Networking
- Network protocols: Rules for communication b/w devices on a network

• Network Interface : Manages connection b/w computer and network

Single Process OS

Only one process executes at a time from ready queue

Batch processing system

① Firstly, user prepares his job using punch cards
② Then he submits the job to the computer operators
③ Operator collects the jobs from different users and sort the jobs into batches with similar needs
④ Then operator submits the batches to the processor one by one
⑤ All the jobs of one batch are executed together

- Priorities cannot be set, if a job comes with some higher priority.
- May lead to starvation (A batch may take more time to complete)
- CPU may become idle in case of i/o operations

Multiprogramming

It increases CPU utilization by keeping multiple jobs (code and data) in the memory so that the CPU always has to execute in case some jobs get busy with i/o.

- Single CPU
- Context switching for processes
- Switch happens when current process goes to wait state.
- CPU idle time reduced.

- **Multitasking** - It is a logical extension of multiprogramming
  - Single CPU
  - Able to run more than one task simultaneously
  - Context switching and time sharing used.
  - Increases responsiveness
  - CPU idle time is reduced further.

- **Multiprocessing OS** - More than 1 CPU in a single computer
  - Increases reliability, 1 CPU fails others can work
  - Better throughput
  - lesser process starvation (if 1 CPU is working on some process, other can be executed on other CPU)

- **Distributed OS** - OS manages many bunch of resources, >=1 CPUs, >=1 memory, >=1 GPUs etc
  - loosely connected autonomous, interconnected computer nodes
  - Collection of independent, networked, communicating and physically separate computational nodes

- **Real time operating system** - A real time operating system is a special purpose operating system which has well defined fixed time constraints. Processing must be done within the defined time limit or system will fail.

  Valued more for how quickly or how predictably it can respond, without buffer delays than for the amount of work it can perform in a given amount of time.

  Ex: Air traffic control systems, ROBOTS etc.

Page No:
Date:

Program: A program is an executable file which contains certain set of instructions written to complete the specific job or operation on your computer

- Its a compiled code, ready to be executed
- Stored in disk

Process: Program under execution. Resides in computers primary memory (RAM).

Thread:
- Single sequence stream within a process
- An independent path of execution in a process
- Light weight process
- Used to achieve parallelism by dividing a process into tasks which are independent path of execution
- Ex: Multiple tabs in a browser
  Text editor

Multi tasking	Multi threading
• The execution of more than one task simultaneously is called multi tasking.	• Process divided into several different ~~threads~~ sub tasks called threads which has its own path of execution.
• Concept of more than 1 processes being context switched	• Concept of more than 1 thread Threads are context switched.
• No. of CPU = 1	• No. of CPU >= 1
• Isolation and memory protection exists. OS must allocate separate memory and resources to each program that CPU is executing.	• No isolation and memory protection, resources are shared among threads of that process. OS allocates memory to a process; multiple threads of that process share same memory & resources allocated to the process.

Page No:
Date:

Thread Scheduling: Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the OS.

Thread Context Switching	Process context switching
• OS saves current state of thread and switches to another thread of same process.	• OS saves current state of process and switches to another process by restoring its state.
• Doesn't include switching of memory address space	• Includes switching of memory address space
• Fast switching	• Slow switching
• CPU cache state is preserved	• CPU's cache state is flushed

Thread Mapping

Threads are of two types:

Threads
- User level
- Kernel level

The CPU executes a user program in two modes
i) User mode ii) Kernel mode

User mode executes most of the instructions.
Few of the restricted instruction can't run in user mode they run on Kernel mode

CPU can't run I/O operations in user mode, they run on Kernel mode.

Page No:
Date:

There are two types of stack → User mode stack
→ Kernel mode stack

The threads which are created inside a single process is the user level thread.
User level thread are created, managed by the user through library functions.

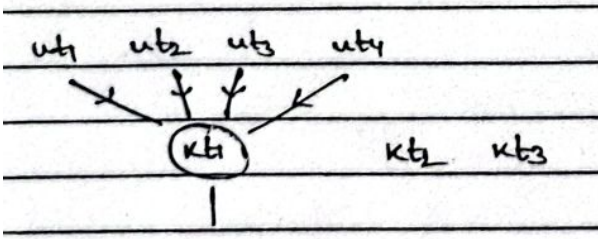
If one of the thread is blocked the remaining thread still also be blocked. Happens in user level thread.
All these user level threads belong to the same address space.
Switching b/w user level threads is faster.

Thread management is easier in user level thread

Switching b/w two kernel level threads can be performed but switching is a single thread can't be performed.

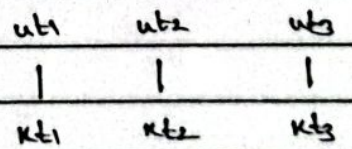
Kernel can create limited threads but user can create any no. of threads.

Mapping → One-one
→ Many-One



(many to one)

It is important because os have limited kernel but so many user level threads

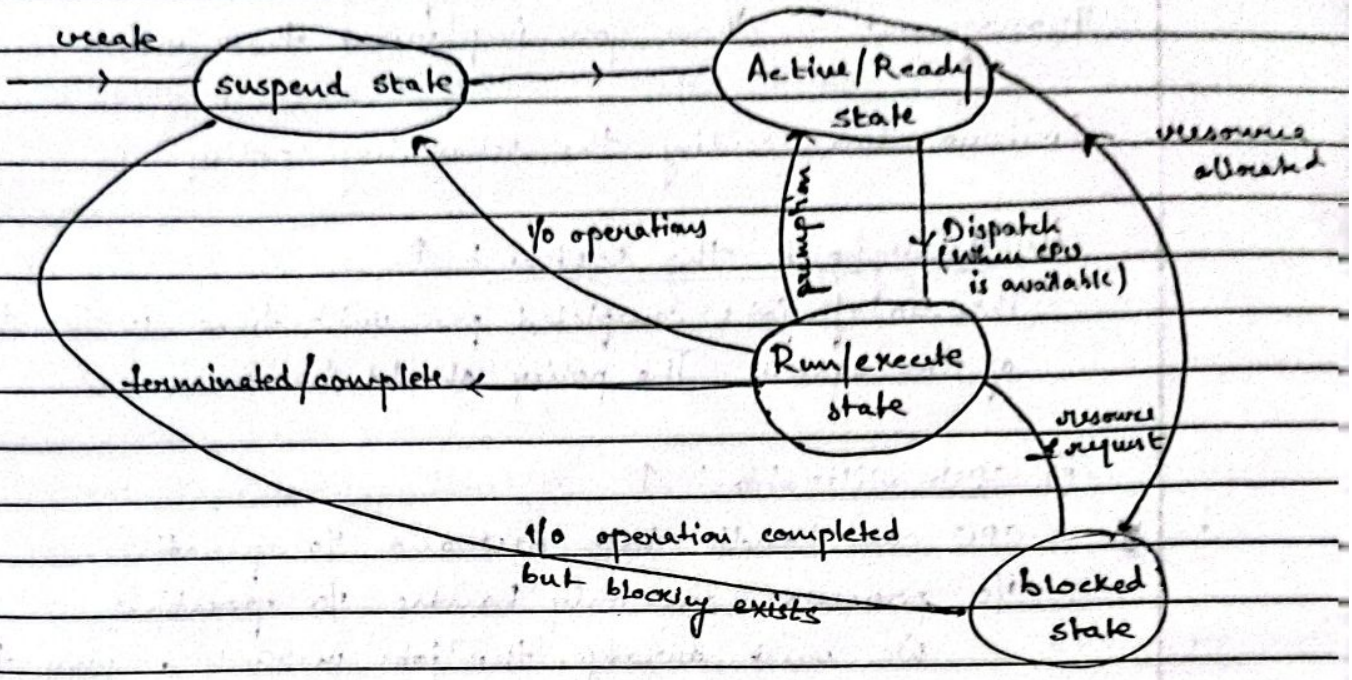


(one to one)

Kernel level threads need more resources so its not that useful.

Page No:
Date:

Process state transition diagram (Snail diagram)



Resource Allocation

- Static
- Dynamic (Min resources to start execution)

Blocked state only appears during dynamic allocation.

A process is in the execution state and requires a resource called X

$P_{ex} \longrightarrow X$

Another process is in the suspended state but have a resource called X

$P_{sup} \longrightarrow X$

Page No:
Date:

- Policy and management

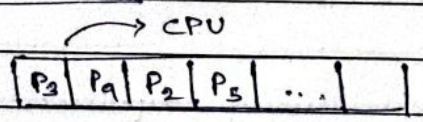
Policy is nothing but a set of rules
Management is how you implement those rules

Criteria for deciding the scheduling policy:

(i) Throughput the system : ↑
The no. of jobs completed per unit time is the throughput of the system. The policy should be increased.

ii) CPU utilization: ↑
CPU can handle both ALU and I/O operations.
I/O processor can only handle I/O operations
We must arrange the jobs in such a way that CPU must be utilised more.

iii) Response time: ↓



Response time is a time a process spends in the active state for the first time being responded by the CPU for execution.

iv) Wait time : ↓
Total time a process waits in the active state until execution.

When a process executes in first attempt
response time = wait time.

v) CPU burst time : ↓
The time CPU takes to execute a process

vi) Turn around time : ↓
Total time for ~~execution~~ completion of executions

Page No:
Date:

Turn around time = CPU burst time + Wait time

* priority should be given → wait time

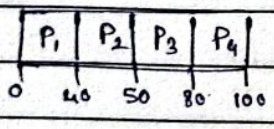
• Average wait time

Avg wait time = Sum of wait time of processes / No. of processes

First come first service

Process that comes first in the active state will go first for execution.

Process	Arrival time	CPU burst time	Wait time	CPU time (max)
P_1	0	40	0	0+40+10+
P_2	0	10	40	30+20
P_3	0	30	50	
P_4	0	20	80	

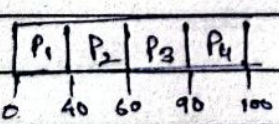


Avg wait time = $\frac{0+40+50+80}{4}$ = 42.5 μsec

FCFS is non preemtive that means it can't be called back before completion.

Process	Arrival time	CPU burst time	Wait time	CPU time (max)
P_1	0	40	0	0+40+20+30
P_2	30	10	60	+10
P_3	20	30	40	
P_4	10	20	30	

Gantt chart



Avg wait time = $\frac{0+60+40+30}{4}$

= 32.5 μsec

Page No:
Date:

Disadvantages of Avg wait time

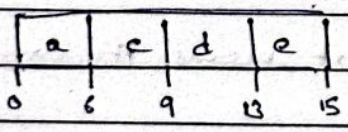
(i) The average wait time would be more.

(ii) If a longer job arrives first all the other small jobs would be waiting for a large amount of time period and as a result the avg wait time will be high.

Shortest job first:

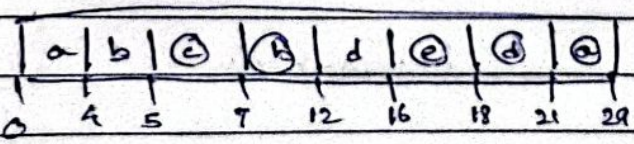
- Non preemtive → Shortest job first
- Preemtive (forcefully taking CPU out of the process without continuing its execution) → Shortest job remaining

Process	at ④	bt ③	wt ①	CPU time ②
a	0	6	0	6 + 3 + 4
b	4	14		
c	5	3	1	
d	8	4		
e	10	2		



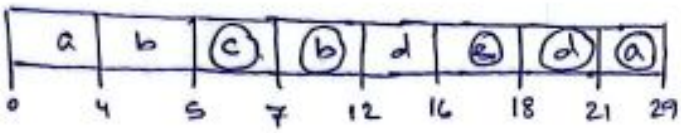
Preemtive

Process	at ④	bt ③	wt ①	CPU time ②
a	~~0~~ 4	~~6~~ 8	0+17	0+4+1+2+5
b	~~4~~ 5	~~14~~ ~~3~~ 0	0+2	+4+2+3+8
c	5	~~3~~ 0	0	
d	~~8~~ 16	~~4~~ ~~2~~ 0	4+2	
e	16	~~2~~ 0	0	



Preemptive

Process	a.t ④	b.t ③	w.t ①	② CPU time
a	0 4	12 8	0+17	0+4+1+2+5
b	4 5	6 ~~5~~ 0	0+2	+4+2+3+8
c	5	2 0	0	
d	8 16	7 ~~3~~ 0	4+2	
e	16	~~2~~ 0	0	



i) Total wait time

ii) Avg wait time

iii) Turn around time

iv) Order of completion of processes

v) Sequence of execution.

vi) Frequency of a process in the runstate.
↳ (How many times a process arrives in the execution state)

vii) ~~When~~ How many context switching are there?
↳ (If a process comes to the active state during its execution)

Disadvantages:

i) If too many small processes arrive the large processes will have to wait for too long. A long long waiting for a process is too high, the phenomena is called ~~start~~ starving.

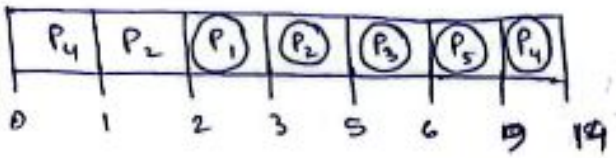
ii) To eliminate starving we use two policies.

a) Round Robin

b) Highest Response Ratio Next

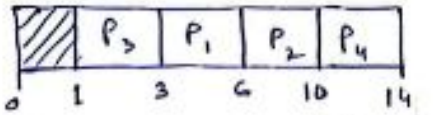
Preemtive

Process	A.t	B.t	Wait time	CPU time
P_1	2	~~1~~ 0	0	
P_2	1	~~3~~ ~~2~~ 0	0+2	
P_3	4	~~1~~ 0	1	
P_4	0	~~6~~ 5	0+9	0+1+1+1+2+1+3+5
P_5	2	3	0+4	



Non-preemtive

Process	A.t	B.t	C.T	Turn around time	W.t	Response time
P_1	1	3	6	5	2	2
P_2	2	4	10	8	4	4
P_3	1	2	3	2	0	0
P_4	4	4	14	10	6	6



Gantt Chart

(TAT) Turn around time = Completion time − Arrival time

Wait time = Turn around time − Burst time

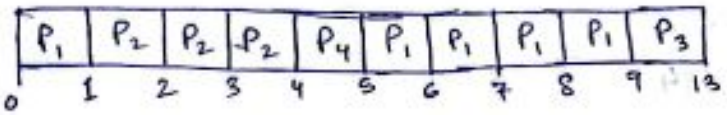
Response time is same as wait time for non-preemtive.

$$\text{Avg TAT} = \frac{5+8+2+10}{4} = 6.25$$

$$\text{Avg wait time} = \frac{2+4+0+6}{4} = 3$$

Shortest Job First for preemtive is called 'Shortest Remaining Time First' (SRTF)

Process	A.T	B.T	C.T	TAT	WT	RT
P_1	0	~~8 4 3 1~~ 0	9	9	4	(0-0)=0
P_2	1	~~8 1~~ 0	4	3	0	(1-1)=0
P_3	2	4 0	13	11	7	(9-2)=7
P_4	4	1 0	5	1	0	(4-4)=0



Grantt Chart

TAT = CT - AT

WT = TAT - BT

RT = First CPU time - AT

Criteria = Burst time

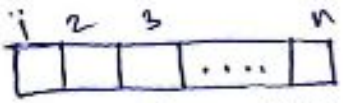
$$\text{Avg TAT} = \frac{9+3+11+1}{4} = 6$$

$$\text{Avg wait time} = \frac{4+0+7+0}{4} = 2.75$$

$$\text{Avg response time} = \frac{0+0+7+0}{4} = 1.75$$

All the processes completed at 13 sec.

Round Robin



AQ (Active State Queue)

q → Each process can be executed in q time

What could be the maximum response time of a proce

For (n-1) no of processes, the total response time

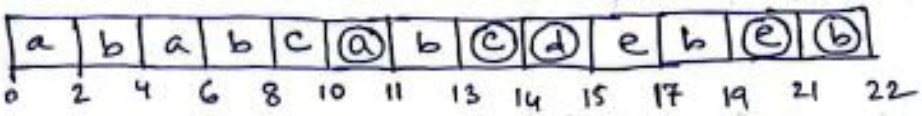
$$(n-1) \times q \rightarrow \max$$

Total maximum time = $(n-1) \times q + q = nq$

Passes #	④ a.t	③ CPU b.t	① w.t	② CPU time
a	0	2	0+2+4	0+2+2+2+2+2+2+1+2+1+1+2+2+2+
b	1	4	1+2+3+4+2	
c	5	1	3+3	
d	10	1	4	
e	15	2	4+2	

time quantam = 2 unit

Purely Preemtive



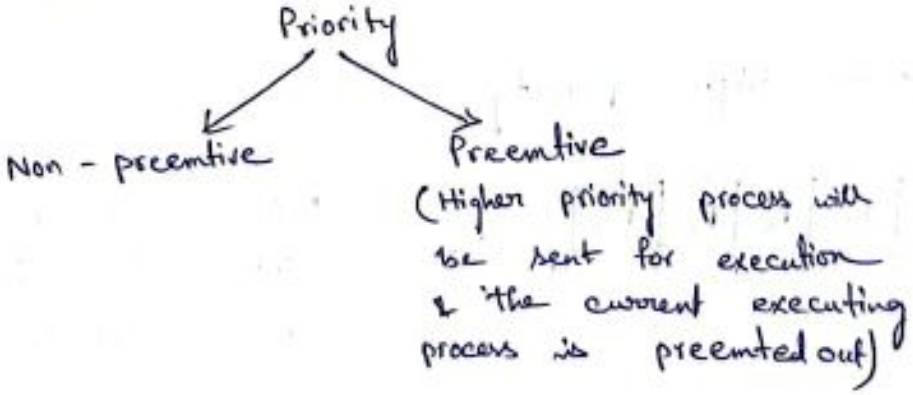
Disadvantages

i) Processes which are Larger are entering more than single time.

ii) Round robin scheduling policy will slow down the execution due to context switching.

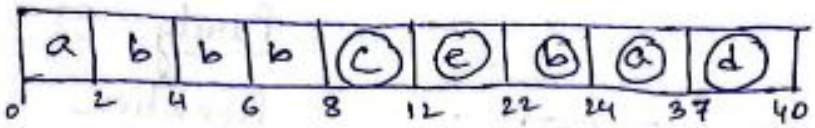
Priority Scheduling

In a real time os priority scheduling is used means which priority is higher will send first.



Non Preemtive

Process #	a.t ④	③ b.t	Priority	w.t ①	② CPU time
a	0 2	~~15~~ 13 0	4	0+22	0+2+2+2+2+4 +10+2+13'+3
b	2 8	8 ~~2~~ 0	3	0+14	
c	8	4 0	1	0	
d	11	3 0	5	26	
e	12	10 0	2	0	

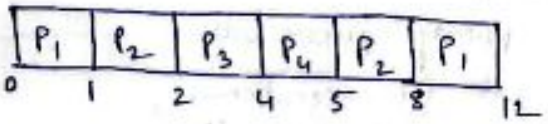


Lower digit indicates higher priority.

Preemtive

Priority	Process	a.t	b.t	c.t	TAT	wt	rt
10	P_1	0	8 ~~4~~ 0	12	12	7	0
20	P_2	1	4 ~~3~~ 0	8	7	3	0
30	P_3	2	~~2~~ 0	4	2	0	0
40	P_4	4	~~1~~ 0	5	1	0	0

Higher no ⟶ Higher priority



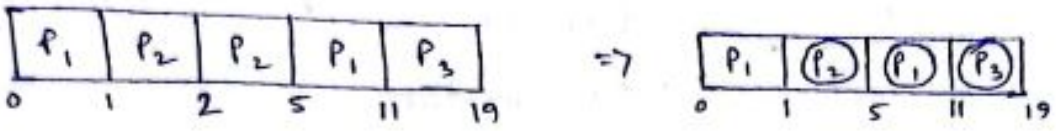
TAT = CT − AT

wt = TAT − BT

RT = First Time CPU − AT

Preemtive SJF → SRTF

Process	a.t	b.t	c.t	TAT	w.t	rt
P_1	0	~~7~~ 6	11	11	4	0
P_2	1	~~4~~ ~~3~~ 0	5	4	0	0
P_3	2	8	19	17	9	9

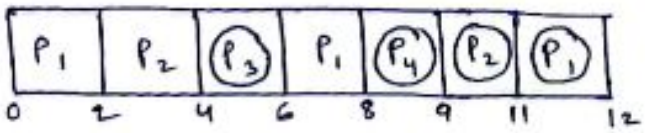


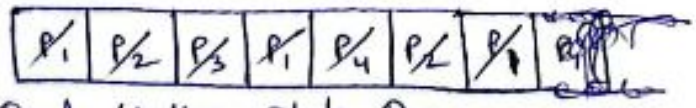
Round Robin

Time Quantum = 2

Process	a.t	b.t	c.t	TAT	w.t	rt
P_1	0	~~8~~ ~~3~~ 1	12	12	7	0
P_2	1	~~4~~ ~~2~~ 0	11	10	6	1
P_3	2	~~2~~ 0	6	4	2	2
P_4	4	~~1~~ 0	9	5	4	4

CPU





Ready / Active State Queue

Process Scheduling Algorithms :

1) First Come First Service → Preemtive
                             → Non-Preemtive

2) Shortest Job First → Preemtive (Shortest Remaining Time Job - SRTJ)
                      → Non-Preemtive (SJF)

3) ~~Rotate~~ Round Robin → Preemtive Only (bcz of TQ)

4) Priority Scheduling → Preemtive
                       → Non-Preemtive

Page No:
Date:

Process Scheduling Algorithms

(i) First come first service → Preemptive / Non preemptive

(ii) Shortest job first → Preemptive (Shortest Remaining Time Job - SRTJ) / Non preemptive (SJF)

(iii) Round Robin → Preemptive only (bez of TQ)

(iv) Priority Scheduling → Preemptive / Non preemptive

(v) Highest response ratio next → Non preemptive

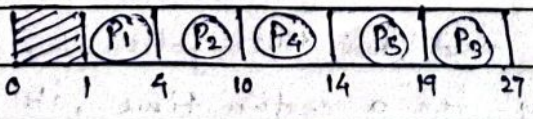
In priority scheduling, starvation occurs to lower priority processes. Aging technique is used to avoid starvation occurs due to priority

Highest Response Ratio Next (HRRN) - (Priority with aging)
- Non preemptive

- Process with highest response ratio will be sent first

$$1 + \frac{\text{wait time}}{\text{CPU burst time}} = \text{Priority of a process (Response ratio)}$$

Process	AT	BT	CT	TAT	WT	RT
P_1	1	3	4	3	0	
P_2	3	6	10	7	1	
P_3	5	8	27	22	14	
P_4	7	4	14	7	3	
P_5	8	5	19	11	6	



at 10

Response ratio for $P_3 = 1 + (10-5)/8 = 13/8 = 1.62$

$P_4 = 1 + (10-7)/4 = 7/4 = 1.75$

$P_5 = 1 + (10-8)/6 = 7/5 = 1.4$

Page No:
Date:

Implementations

(i) **First Come First Serve** : Will have a circular queue where new process will be added at the end and processes are ready for execution are extracted from begining.

(ii) **Shortest Job First** : Will have a sorting algo with 'nlogn' time complexity. After completion of each process or after certain amount of time, will have to run the sorting algo for repeatative nos. So the total running time will be huge.

So SJF is implemented using heap where processes can come (insertion) and go (deletion) in logn time (much more faster than 'nlogn' sort algo)

(iii) **Round Robin**: Will have a counter that will count downward according to the time quantum given. When it comes to 0 the currently executing process is sent to the back of the circular queue and a new process is sent for execution. Again the counter will start counting

- **Multilevel Priority Queue**: Real Scheduler

Real time and system jobs are very important and smaller. We can apply first come first serve policy.

Because the job sizes are unequal / uneven we go for Round Robin.

For batch jobs we use first come first serve
No scheduling policy is used for NULL jobs
If a job isn't responding for a certain time, its priority will be increased.
After executing for certain time, if the job isn't completed the priority gets decreased

Page No:
Date:

System Calls

When a user uses an application or an API basically the user is in User mode. But if we need to use the functionalities of OS we need to swith to the Kernel mode

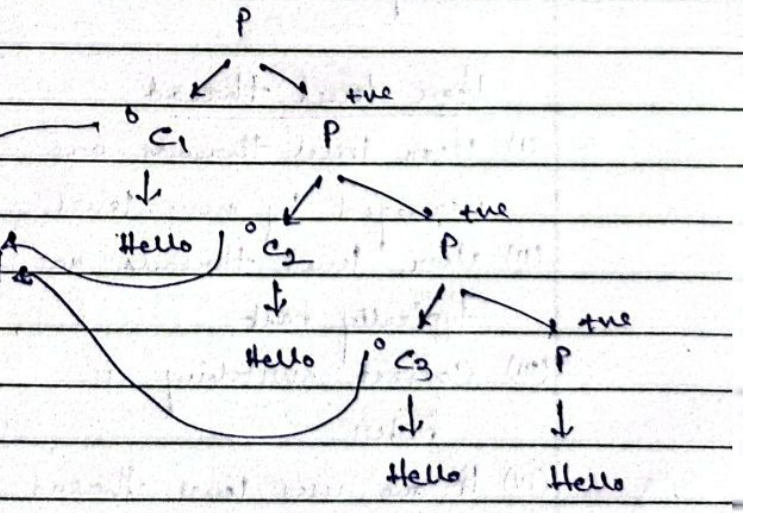
System call is the way of switching from user mode to Kernel mode to access the functionalities of OS. User mode doesn't have the direct access to the hardware thats why we need to switch to the Kernal mode to access via OS

System Calls
- File related => Open(), Read(), Write(), Close(), Create()
- Device related => Read, Write, Reposition, ioctl, fcntl
- Information => get pid, attributes, get system time and data
- Process control => load, execute, abort, fork, wait, signal, allocate
- Communication => pipe(), create/delete connection, Shmget()

Fork: Suppose there is a program that can be referred as a parent program. If we use 'fork' inside the parent program it will create a child program (basically clone of parent program)

Fork → 0 (child)
Fork → 1 (parent)

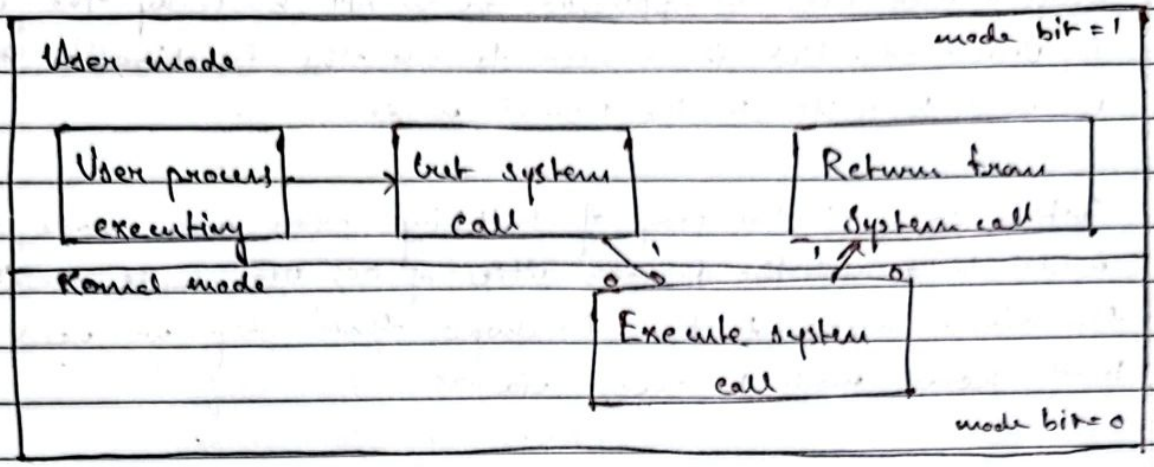
```
#include <stdio.h>
# include <unistd.h>
int main()
{
if (fork() && fork())
  fork();
printf("Hello");
return 0;
}
```



No. of times Hello will be printed

Page No:
Date:

User mode v/s Kernel mode



Process	Threads
(i) System called involved in process	(i) There is no system call involved
(ii) OS treats different process differently	(ii) All user level threads treated as single task for OS.
(iii) Different process have different copies of data, code, files.	(iii) Threads share same copy of code and data
(iv) Context switching is slower	(iv) Context switching is faster
(v) Blocking a process will not block another.	(v) Block entire process
(vi) Independent	(vi) Interdependent

User level thread	Kernel level thread
(i) User level threads are managed by user level	(i) Kernel level threads are managed by OS
(ii) User level threads are typically fast	(ii) Kernel level threads are slower than user level
(iii) Context switching is faster	(iii) Context switching is slower
(iv) If one user level thread performs blocking operation entire process gets blocked	(iv) If one Kernel level thread is blocked, no effect on other Kernels

Page No:
Date:

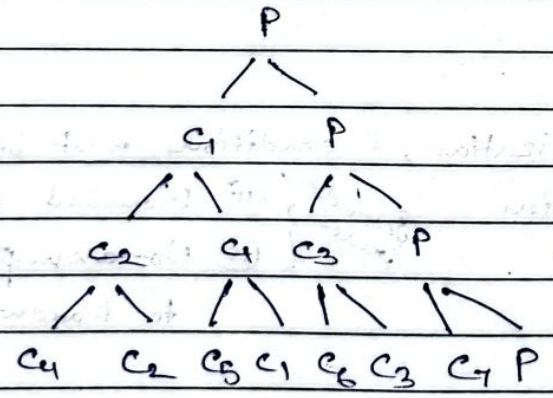
Context Switching Time

Process > Kernel level thread > User level thread

(i) PCB holds information about a process
(ii) The exe code is loaded into main memory
(iii) The stack and heap area is also assigned

(i) fork() is a system call. (Create()) This system call creates a child process which is a replica of the parent.
(ii) fork() returns 0 to the child process and process id of the child process to the parent.
(iii) The execution to the child process starts next to the fork() command.
(iv) In UNIX OS every hardware device is considered as a file
(v) Every process will have a process ID (in PCB) → starts from 1
(vi) Parent returns 0 to the child

P_id for parent ⇒ P_id returned by child
P_id for child ⇒ 0 returned by parent



child = 7
Total = 8

No. of childs = $2^n - 1$

```
int a, Pid
for i=1 to 3
do {
  fork();
}
```

≡ fork() fork() fork()

Process Synchronization

Process Synchronization is the coordination of execution of multiple processes in a multi process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.

Process Synchronization

Independent	Cooperative
The execution of one process doesn't affect the execution of other processes.	A process that can affect or be affected by other processes executing in the system.

Critical Section Problem

A critical section is a code segment that can be accessed by only one process at a time. It contains shared variables that need to be synchronized to maintain the consistency of data variables.

To achieve synchronization, 4 conditions must be satisfied:

Primary rules: (I) Mutual exclusion, (II) Progress

Secondary rules: (III) Bounded wait, (IV) No assumption related to hardware & speed

- Mutual exclusion: If process P_i is executing in its critical section, then no other processes can be executing in their critical section.

- Progress: If no process is executing in its critical section and some processes wish to enter their critical section, then only those processes that aren't executing in their remainder sections can participate in deciding which

will enter the critical section next and this selection cannot be postponed indefinitely.

- Bounded waiting : There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Use of critical sections in a program can cause a number of issues including :

(i) Deadlock
(ii) Starvation
(iii) Overhead

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

```
do {
    flag = 1;
    while (flag);      // entry section
                       // critical section
    if (!flag)
                       // remainder section
} while (true);
```

Page No:
Date:

Classic S/W soln: Peterson's Solution

- Restricted to two processes
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted (not true for modern processors)
- The two threads share two variables:
  - int turn;
  - Boolean flag[2];
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready

Algorithm for process Pi

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
        CRITICAL SECTION
    flag[i] = FALSE;
        REMAINDER SECTION
} while (TRUE);
```

① Mutual exclusion because only way thread enter critical section when flag[j] == FALSE or turn == TRUE
② Only way to enter section is by flipping flag[] inside loop
③ turn = j allows the other thread to make progress

Solution using TestAndSet

- Defination of TestAndSet

```
boolean TestAndSet (boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
    }
```

- Shared boolean variable lock., initialized to false
- Solution

```
do {
  while (TestAndSet (&lock))
      ; /* do nothing
      // critical section
  lock = FALSE;
      // remainder section
  } while (TRUE);
```

Solution using Swap

- Defination of Swap

```
void swap (boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
    }
```

- Shared boolean variable lock initialized to FALSE ; Each process has a local Boolean variable Key
- Solution :

```
do {
    Key = TRUE;
    while (Key == TRUE)
        Swap (&lock, &Key);
            // critical section
    Lock = FALSE;
            // remainder section
    } while (TRUE);
```

Solution with TestAndSet and bounded wait

- boolean waiting [n]; boolean lock; initialized to false

Pi can enter critical section iff waiting [i] == false or key == false

```
do {
    waiting [i] = TRUE;
    Key = TRUE;
    while (waiting [i] && Key)
        Key = TestAndSet (&lock);
    waiting [i] = FALSE;
        // critical section
    j = (i+1) % n;
    while ((j != i) && !waiting [j])
        j = (j+1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting [j] = FALSE;
        // remainder section
} while (TRUE);
```

Semaphore Synchronization ~~Techniq~~ Primitive

- Test And Set are hard to program for end users
- Introduce a simple function called semaphore
    - Semaphore is an integer S
    - Only legal operations on S are:
        - Wait() [atomic] - if S > 0, decrement S else loop
        - Signal () [atomic] - increment S

```
• wait(S){
    while S <= 0
        ; // no-op
        S--;
    }
• signal (S) {
        S++;
    }
```

- Counting (S: is unrestricted), binary (mutex lock) (S: 0, 1)

Page No:
Date:

Semaphore

Binary Semaphore
This is also known as a mutex lock. It can only have two values - 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes

Counting Semaphore
Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Advantages of Process Synchronization
- Ensures data consistency and integrity
- Avoids race conditions
- Prevents inconsistent data due to concurrent access
- Supports efficient and effective use of shared resources

Disadvantages of process synchronization
- Adds overhead to the system
- Can lead to performance degradation.
- Increase in complexity of the system.
- Can cause deadlocks if not implemented properly.

Mutex Locks
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first acquire() a lock then release() the lock.
  - Boolean variable indicating if lock is available or not
- Calls to acquire() and release() must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires busy waiting
  - This lock therefore is called Spinlock

```
• acquire () {
      while (!available)
        ; /* busy wait */
        available = false;
        }
• release () {
      available = true;
      }
• do {
      acquire lock()
            // critical section
      release lock()
            // remainder section
  } while (true);
```

Semaphore implementation with no Busy waiting

```
typedef struct {
    int value;
    struct process * list;
    } semaphore;

wait (semaphore *S) {
    S→ value--;
    if (S→ value < 0) {
        add this process to S→ list;
        block();
        }
    }

signal (semaphore *S) {
    S→ value++;
    if (S→ value <= 0) {
        remove a process P from S→ list;
        wakeup (P);
        }
    }
```

Page No:
Date:

Producer Consumer Problem

Suppose we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer counter that keeps track of the number of full buffers. Initially counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
    /* produce an item in next produced */
    while (counter == BUFFER_SIZE);
        /* do nothing */
    buffer [in] = next_produced;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

Consumer

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer [out];
    out = (out+1) % BUFFER-SIZE;
        counter--;
    /* consume the item in next consumed */
}
```

Reader - Writers Problem

- A dataset is shared among a number of concurrent processes
  - Readers : Only read the dataset ; don't perform any updates
  - Writers : Can both read and write
- Problem - allow multiple readers to read at the same time.
  - Only one single writer can access the shared data at the shared time.
- Several variation of how readers and writers are considered all involve some form of priorities
- Shared data - Data set
  - Semaphore rw_mutex initialized to 1
  - Semaphore mutex initialized to 1
  - Integer read_count initialized to 0

Writer :

```
do {
    wait (rw_mutex);
    ...
    /* writing is performed */
    ...
    signal (rw_mutex);
} while (true);
```

Writer :

```
do {
    wait (mutex);
    read_count ++;
    if (read_count == 1)
        wait (rw_mutex);
    signal (mutex);
    ...
    /* reading is performed */
    ...
    wait (mutex);
    read_count --;
    if (read_count == 0)
        signal (rw_mutex);
    signal (mutex);
} while (true);
```

Reader - Writer Problem Variations

First variation : no reader kept waiting unless writer has permission to use shared object
Second variation : once writer is ready, it performs the write ASAP
Both may have starvation leading to even more variations
Problem is solved on some systems by kernel providing reader-writer locks

DINING PHILOSOPHERS PROBLEM

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbours, occassionally try to pick 2 chopsticks (one at a time) to eat from bowl.
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

```
do {
    wait (chopstick [i]);
    wait (chopstick [(i+1) % 5]);
        // eat
    signal (chopstick [i]);
    signal (chopstick [(i+1) % 5]);
        // think
    } while (TRUE);
```

Deadlock Handling

- Allow atmost 4 philosophers to be sitting simultaneously at table
- Allow a philosopher to pick up the forks only if both are available (picking must be done in critical section)
- Use an assymetric solution -- an odd numbered philosophers picks up first the left chopstick and then right chopstick. Even numbered philosopher picks up first the right chopstick and then left chopstick

Advantages of semaphores

- Simple and effective mechanism for process synchronization
- Supports coordination b/w multiple processes
- Can be used to implement critical sections in program
- Used to avoid race conditions

Disadvantages of semaphores

- It can lead to performance degradation due to overhead associated with wait and signal operations.
- Can result in deadlock if used incorrectly
- It can cause performance issues in a program if not used properly.
- It can be prone to race conditions and other synchronization problems if not properly used

Monitors

Monitors are used for process synchronization. Monitors are defined as construct of programming language, which helps in controlling shared data access

* Characteristics

① Inside monitor, we can execute one process at a time
② A group of procedures and condition variables that are merged together in special type of module.
③ Offer high level of synchronization

Syntax

```
Monitor demo // Name of the monitor
{
variables ;
condition variables ;
procedure p1 {.... }
procedure p2 {.... }
}
```

Monitor	Semaphore
(i) We can use condition variables only in monitor	(i) In semaphore, we can use condition variables anywhere in the program but we cannot use condition variables in a semaphore.
(ii) Wait always block the caller	(ii) Wait doesn't always block the caller.
(iii) Monitors comprised of shared variables and procedures which operate the shared variable.	(iii) The semaphore's values means the number of shared resources that are present in the system.
(iv) Condition variables present in monitor	(iv) Condition variables not present in semaphore

Dining Philosophers problem using Monitor

"Five philosophers sit around a circular table. Each philosopher spends his life alternatively thinking and eating. In the center of a table is a large plate of spaghetti. A philosopher needs two forks to ~~help~~ eat a helping of spaghetti. Unfortunately, as philosophy is not as well paid as computing, the philosophers can only afford five forks. One fork is placed between each pair of philosopher and they agree that each will only use the fork to his immediate right and left."

↳ N philosophers seated around a circular table
- There is one fork b/w each philosopher
- To eat, a philosopher ~~must pick up their~~ must pick up their two nearest forks.
- A philosopher must pick up one fork at a time, not both at ones.

Page No:
Date:

Possibility of Deadlock

If philosophers take one fork at a time, taking a fork from the left and then one from the right, there is a danger of deadlock. This possibility of deadlock means that any solution to the problem must include some provision for preventing or otherwise dealing with deadlocks.

Possibility of Starvation

If philosophers take two forks at a time, there is a possibility of starvation. Philosophers P_2 and P_5 and P1 & P_3 can alternate in a way that starves out philosopher P_4. This possibility of starvation means that any solution to the problem must include some provision for preventing starvation

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
  enum (THINKING, HUNGRY, EATING) state [5];
  condition self [5];

  void pickup (int i) {
        state [i] = HUNGRY;
        test (i);
        if (state [i] != EATING) self [i].wait;
  }

  void pushdown (int i) {
        state [i] = THINKING;
              // test left and right neighbours
        test ((i+4) %5);
        test ((i+1) %5);
  }
```

```
void test (int i) {
    if ((state[(i+4) %. 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i+1) %. 5] != EATING)) {
            state[i] = EATING;
        self[i].signal();
        }
    }

initialization_code() {
    for (int i=0; i<5; i++)
    state[i] = THINKING;
    }
}
```