Operating System

Operating System works as a resource manager to a computer system.

Resources : i) CPU / GPU

ii) Memory

iii) Input & Output devices (I/O)

CPU Manager : Decides which task will be given the first preference. It arranges the programs (process) in a particular order by the CPU for execution. It is also called as CPU Scheduler.

Memory Manager : Memory Manager keeps track of information (Used and free) of the main memory. After that it'll do allocation & deallocation of the programs.

I/O Manager : It works as a controller which controls all the I/O devices connected to the system.

OS will work as an interfacing b/w a human being and a computer (system).

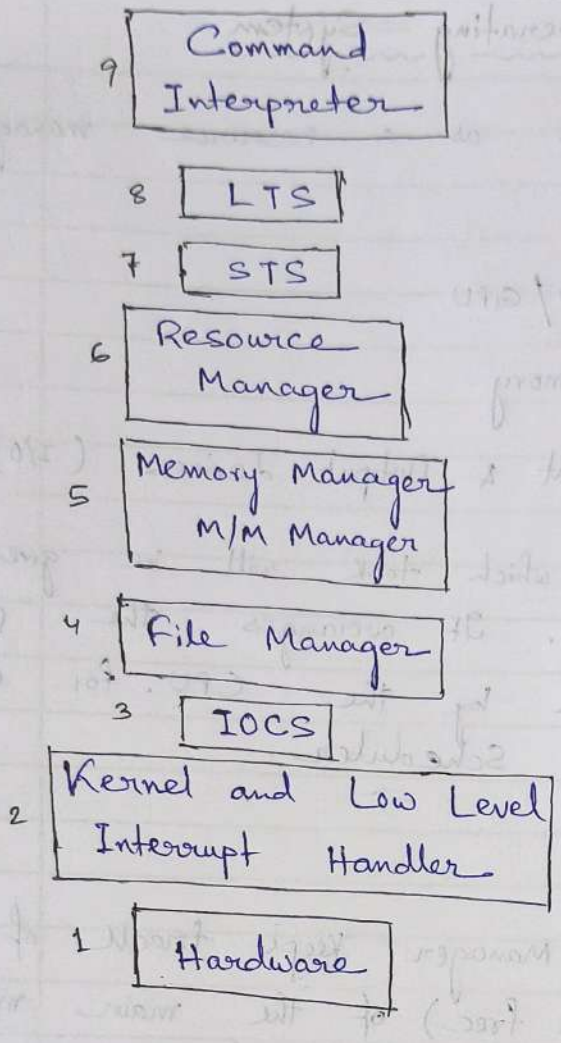
Accountant :

Guardian : OS provides protection to those files that shouldn't be deleted.

Coordinator :

Server : OS provides different types of services like copying, pasting etc.

Resource Utilization Maximizer :



2) Kernel is the integral part of the OS. Mostly dealing with ~~processes~~ processes and low level interrupt handlers.

3) Input output control system which governs all types of I/O operations.

7) STS → Short Term Scheduler or CPU Managers prepare an order in which CPU will dispatch for execution.

8) Long Term Scheduler.

Before execution of a program a process needs to be created. A process control block holds information about a process during it's execution / through out it's execution.

Program in execution with a unique set of data is called as a process.

LTS is responsible for creating a process.
Once process is created it will go to STS.

1) Process Synchronisation
2) Protection & Security
} Topic for OS

History of Operating System :

Different development stages of Operating System.

a) Batch OS :

i) Interactive Program (During execution you need to interact with computer)
ii) Non-Interative Program (Keyboard, mouse)
(Doesn't have such type I/O statements)
This type of programs collect & store data from the disk not from the user.

abc.bat
{ a.exe
b.exe
c.exe

Without ~~After~~ entering all the three programs manually, you've to enter the batch file once and all the three programs will run automatically.

In batch OS one program can be there in the memory and programs are executed one after another in a sequential order are mentioned or given in the batch file.

Advantage :

i) User time can be saved.
ii)

b) Multiprogramming OS :

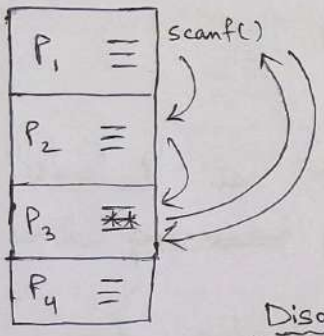
i) Multiple programs reside in the main memory at a time.

ii) CPU switches over to another program if the current program completes the execution or the current program encounters an I/O operation.

iii) Multiple programs are executed simultaneously (concurrently in reality) in a uniprocessor system is named as Multiprogramming OS.

Three types of execution :

- Sequential (One after another execution)
- Parallel (Simultaneous execution at the time)
- Concurrent (In case of multiprogramming OS)



As I/O operation is encountered, CPU moves to P_2. After completion of P_2 it moves to P_3 and at execution time as it receives input moves to P_1 completes it and moves to statement in P_3 till which it's been executed before.

Disadvantages :

Size of processes are uneven.

P_1	P_2	P_3	P_4
20 min	30 min	2 sec	4 sec

c) Time Sharing OS :

Every program is given a fixed time.

Suppose the time is 2 sec

P_1	P_2	P_3	P_4
21 min	30 min	2 sec	4 sec
2 sec	2 sec	2 sec	2 sec

So if a job is very large then larger users would need many cycles to go through the CPU. It affects large file users. Special case of multiprogramming.

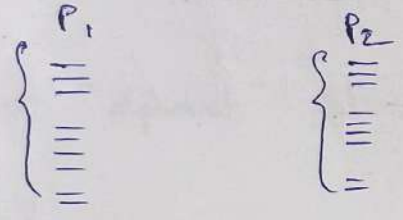
d) Real Time OS:

Real time jobs are very shoot & quick. Jobs are executed based on their property. Real time jobs should be prioritised over online jobs as well as offline jobs.

e) Multiprocessing OS:

Multi processes are executed simultaneously on reality in a multiprocessor system. → Distributed OS
→ Networking OS

f) Multithreaded OS:



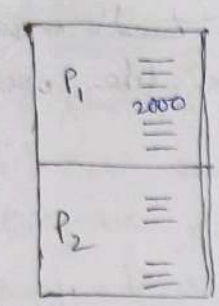
- Higher priority programs arrive.
- I/O scenario.
- Time quantum is finished.

Context switching has these reasons means execution of program is switched to the next program.

Every program when it starts it's execution is assigned with registers, program counters, stack (memory, storage) [R_0 to R_{32}]

Some variables are kept in registers which are visited a lot. When we switch from one program to other we will store registers in our stack and when return back we will pop.

Program Counter (PC) holds address of next program. When we switch to it and then as we shift to next program address till which the previous program was executed is stored.



Suppose till 2000 is executed the address of next program is held by PC and as next code starts to execute, PC saves the next address of first program i.e. (2004) from where it needs to start executing once it returns back to the first program.

---

whoami → Prints the username of the current user.

who → Prints the list of all users that are currently logged into the system.

man who → Used to display the user manual of any command that we can run on the terminal.

cat > "file name" → To create a file

rm "file name" → To delete a file.

ctrl + d → To save the file

cat first → To print whatever written inside.

ls → To list the contents of a directory

ls -l →

-r-x-rw- 2 → file

drwxr-xr-x 2 → Directory

head "file name" → To display top to bottom only first 10 no of lines.

head -2 "file name" → To display the no of lines entered as an integer.

tail "file name" → Bottom to top prints 10 numbers

tail -3 "file name" → Bottom to top 3 integers.

first	head first	head -2 first	tail first	tail -2 first
a	a	a	c	k
b	b	b	d	l
c	c		e	
d	d		f	
e	e		g	
f	f		h	
g	g		i	
h	h		j	
i	i		k	
j	j		l	
k				
l				

cat "filename 1" > "filename 2" ⟶ Copies all the contents of filename 1 to filename 2

grep "alphabet" filename ⟶ Highlights wherever the passed alphabet or string is present inside the file.

touch "filename" ⟶ Creates an empty file.

Suppose first contains alphabet a to l
Now create a blank file ⟶ touch second
Now copy the contents of first to second ⟶ cat first > second
Now to check if copied ⟶ cat second.

cp "filename 1" "filename 2"
↳ Copies all the contents of filename 1 to filename 2.
If filename 2 doesn't exist at first it'll be created automatically.

mkdir dir1/dir2 → To create directories with subdirectories

mkdir "dir1" "dir2" → Adding space b/w two directory names will create that no of directories under your current directory.

pwd → To show the current working directory

mv "dir1" "dir2" → Copies all the files from dir1 to dir2.

rm → To delete a file.

rmdir → To delete a directory.

'|' (Pipeline)

command1 | command2

The left part will be redirected to right part (next command) after executing the left part command.

head -3 first | tail -1

This will work as the i/p in the second command

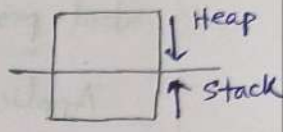
final o/p

cut -b -3 "filename" → Prints upto the 3rd character of each line.

Process Creation :

i) A PCB (Process Control Block) holding information about a process is created in the main memory.

ii) A memory space is assigned for
a) Code
b) data
c) stack
d) Heap



iii) The code is loaded to the main memory.

iv) The stack must be initialized. (stack basically holds local variables and parameters.)

v) Some special purpose and general purpose registers are also stored in the stack.

The time which the OS spends during Context Switching is known as Overhead Time.

Q Can we minimize the context switching time ?

*) An unique set of GPR and SPR is called as Thread of Execution or Control (TOC)

Having multiple sets of TOC (having hyperthreaded technology) will result in reducing the Overhead Time.

Multithreaded OS :

Area of a circle is found by a program.

i) Thread creation time is less than process creation.

ii) Switching b/w threads take less time than process switching
↳ (for similar threads)

iii) Thread also shares the files as well as the code and requires less resources compared to a process.

iv) First creation time will be same as process then it'll take less time for similar threads.

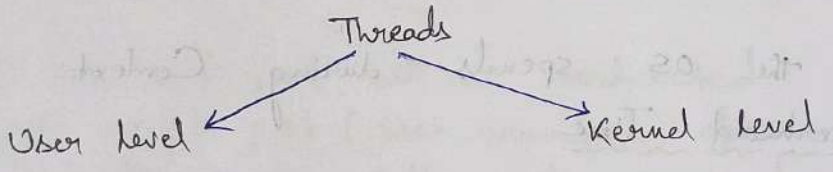
Multiple threads run for same application (by multiple users) : Client server architechture.

Word processing Application
- Fetch data
- Update display
- Spell check
- Sentence check

The whole application process is a process and each functions are thread.

Thread Mapping

Threads are of two types.



The CPU executes a user program ~~into~~ in two modes.
1) User mode 2) 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 in Kernel mode.

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 threads are created, managed by the user through library functions.

If one of the thread is blocked (by scanf) the remaining thread will also be blocked. Happens in user level thread.

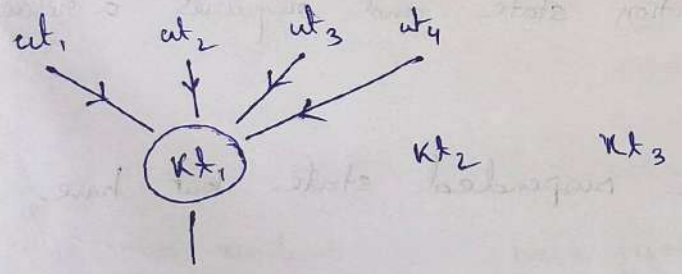
All these user level threads belong to same address space. Switching b/w user level threads is faster.

Thread management is easier in user level thread.

Switching b/w two kernel level threads can be performed (if scanf is encountered) but switching in a single thread can't be performed.

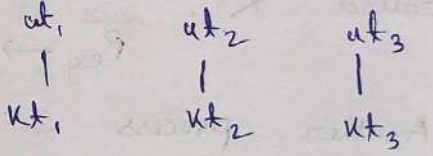
Kernel can create limited threads but user can create any no of threads.

Mapping → One – One
Mapping → Many – One



Many to One

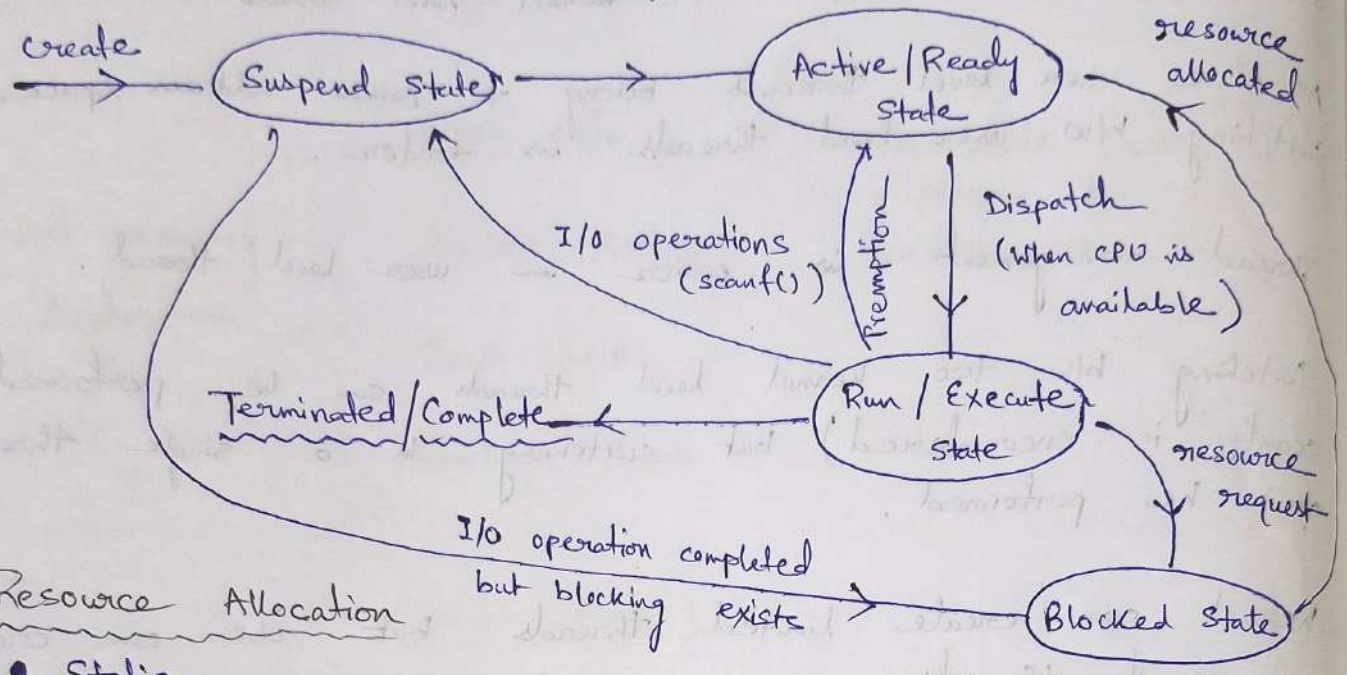
(It is imp bez OS have limited kernel but so many user level threads)



One to One

(Kernel level threads need more resources so it's not that useful)

Process State Transition Diagram : (Small 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.

$$P_{sup} \longrightarrow X$$

echo "..." → To print something

read a → "..." → To take user input

echo $a → Will print the value stored inside a

Shellscript is a collection of commands.

read a b c → To take multiple user inputs

gedit example.sh → To create new file (.sh → extension)

```
echo Enter two numbers :
read a b
echo Printing the user input :
echo $a $b
```

O/P

```
Enter two numbers :
12 2
Printing the user input :
12 2
```

To run → sh example.sh

```
c='expr $a + $b'
```

no space

space

```
echo Enter two numbers :
read a b
echo Printing the sum :
c='expr $a + $b'
echo $c
```

O/P

```
Enter two numbers :
12 2
Printing the sum :
14
```

back quote(`) → To assign output from system commands to variables

---

* → Multiplication

/ → Division

```
echo Enter two numbers :
read a b
echo Multiplication is :
c='expr $a \* $b'
echo $c
d='expr $a / $b'
echo $d
echo Division is :
```

O/P

```
Enter two numbers :
12 2
Multiplication is :
24
Division is :
6
```

Q. Write a shell script to display hour, minute, seconds from user inputed seconds ?

Q. Write a shell script to find out how many 100, 50, 20 rupees are required for a user entered amount ?

> -gt	&& -a	if [cond^n]	if (—)
>= -ge	|| -o	then	{
< -lt	(logical)	≅ equivalent	
<= -le		fi	}
== -eq			
!= -ne			
(relational)			

```
if [ $a -gt $b]
then
   echo greater is $a
fi
else
   echo greater is $b
```

```
if [    ]
then
elif [    ]
then
:
fi
```

⟷ Same

```
if [    ]
then
else if [    ]
Then
fi
else if [    ]
then
fi
:
```

Write a shell script to find the greatest among 3 numbers using logical operator, nested if, else-if ladder.

Write a shell script to check whether a input no is even or odd.

Write a shell script to find out reverse of a 3 digit no.

Write a shell script to check a 3 digit no is palindrome or not.

Policy and Management

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

Criteria for deciding the scheduling policy:

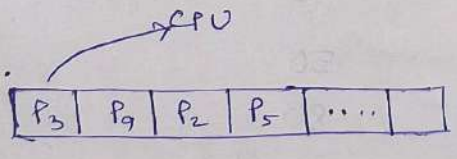
i) Throughput of the system: ↑

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 utilized more.

iii) Response time: ↓



Active State Queue

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 : ↓

The total time for completion of executions.

Wait { 10:00 → Reach
10:10 → Enter

break { 10:30 → Answering
3:00 → Reach again

Wait { 3:30 → Enter
3:35 → Answering done

Turn around time = CPU burst time + Wait time

Priority should be given → Wait time

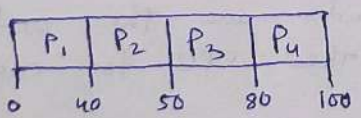
Average Wait Time :

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 (Time in μsec)	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	



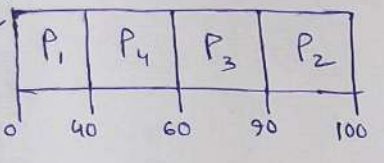
Avg wait time = $\frac{0 + 40 + 50 + 80}{4}$

= 42.5 μsec

FCFS is non-premtive 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 + 10
P_2	30	10	60	
P_3	20	30	40	
P_4	10	20	30	

Gantt chart



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

$$= 32.5 \text{ msec}$$

Disadvantages of Avg Wait Time

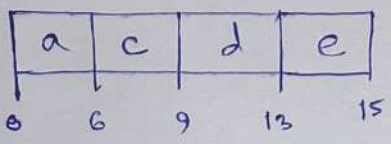
i) The avg wait time would be more.

ii) If a larger 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 :

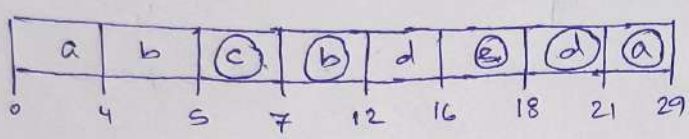
- Non-preemtive → Shortest Job First
- Preemtive (Forcefully taking CPU out of the process without continuing it's execution) → Shortest Job Remaining

Process :	a.t ④	b.t ③	w.t ①	CPU time ②
a	0	~~6~~ 0	0	6 + ~~3~~ 3 + 4
b	4	14		
c	5	~~3~~ 0	1	
d	8	4		
e	10	2		



Preemtive

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	

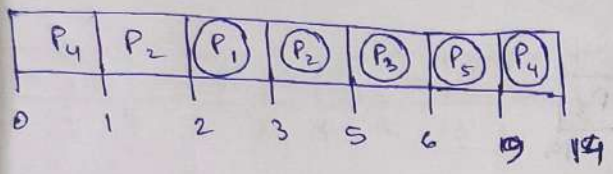


i) Total wait time
ii) Avg wait time
iii) Turn around time
iv) Order of completion of processes
v) Sequence of execution.
vi) Frequency of a process in the runstate.
↳ (How many times a process arrives in the execution state)
vii) ~~When~~ How many context switching are there?
↳ (If a process comes to 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

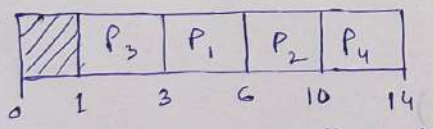
Preemtive

Process	A.t	B.t	Wait time	CPU time
P_1	2	~~1~~ 0	0	
P_2	1	~~3~~ ~~1~~ 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



Grantt Chart

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

Wait time = Turn around time − Burst time

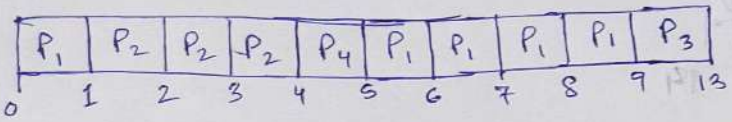
Response time is same as wait time for non-preemtive.

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

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

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	~~6 4 3 2 1~~ 0	9	9	4	(0-0)=0
P_2	1	~~3 2 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



Gantt Chart

TAT = CT - AT

WT = TAT - BT

RT = First CPU time - AT

Criteria = Burst time

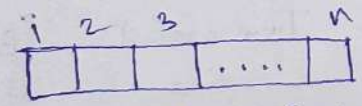
$$\text{Avg TAT} = \frac{9+3+11+1}{4} = 6$$

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

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

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 process

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

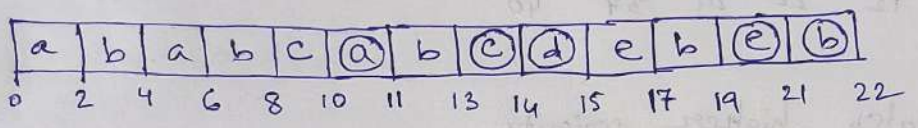
$$(n-1) \times q \longrightarrow \max$$

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	6	0+2+4	0+2+2+2+2+2+1+2+1+1+2+2+2+1
b	1	13	1+2+3+4+2	
c	5	10	3+3	
d	10	4	4	
e	11	17	4+2	

time quantum = 2 unit

Purely Preemtive



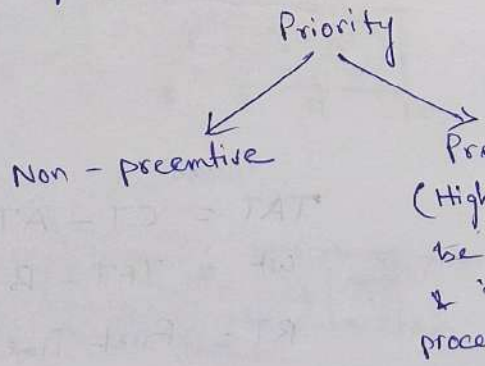
Disadvantages

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.

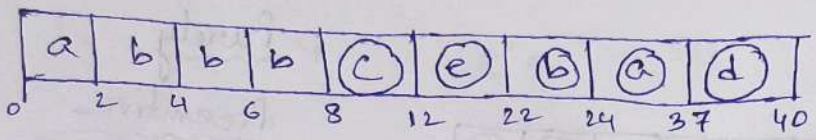
Priority Scheduling

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	

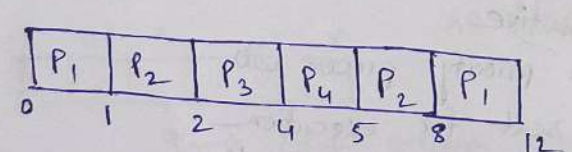


Lower digit indicates higher priority.

Preemtive

Priority	Process	a.t	b.t	c.t	TAT	wt	rt
10	P_1	0	~~8~~ 7 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



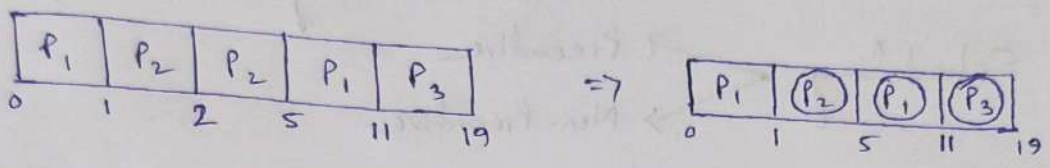
TAT = CT − AT

wt = TAT − BT

RT = First Time CPU − AT

Preemtive SJF → SRTF

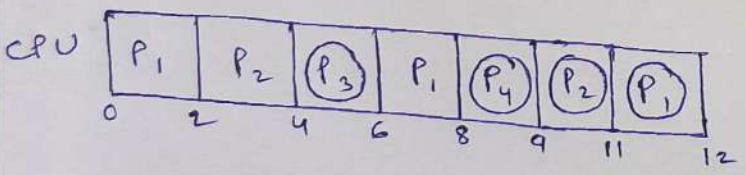
Process	a.t	b.t	c.t	TAT	wt	rt
P_1	0	~~7~~ 6	11	11	4	0
P_2	1	~~4~~ ~~3~~ 0	5	4	0	0
P_3	2	8	19	17	9	9

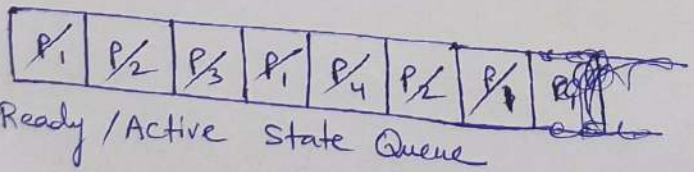


Round Robin

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





Process Scheduling Algorithms :

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

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

iii) ~~Rotata~~ Round Robin → Preemtive Only (bcz of TQ)

iv) Priority Scheduling → Preemtive / Non-Preemtive

v) Highest Response Ratio Next → Non-Preemtive

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

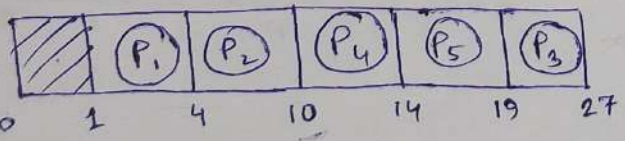
(HRRN)

Highest Reponse Ratio Next (Priority with aging) → Non-Preemtive

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	(1-1) = 0
P_2	3	6	10	7	1	(4-3) = 1
P_3	5	8	27	22	14	(19-5) = 14
P_4	7	4	14	7	3	(10-7) = 3
P_5	8	5	19	11	6	(14-8) = 6



You've to calculate RR after completion of every process. Highest RR → Will be sent first

at 10

Response ratio for $P_3 = 1 + \frac{(10-5)}{8} = \frac{13}{8} = 1.62$

" " " $P_4 = 1 + \frac{(10-7)}{4} = \frac{7}{4} = 1.75$

" " " $P_5 = 1 + \frac{(10-8)}{5} = \frac{7}{5} = 1.4$

at 14

" " " $P_3 = 1 + \frac{(14-5)}{8} = \frac{17}{8} = 2.125$

" " " $P_5 = 1 + \frac{(14-8)}{5} = \frac{11}{5} = 2.2$

Implementations :

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

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'll count downwards according to the time quantam 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 Schedular

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.

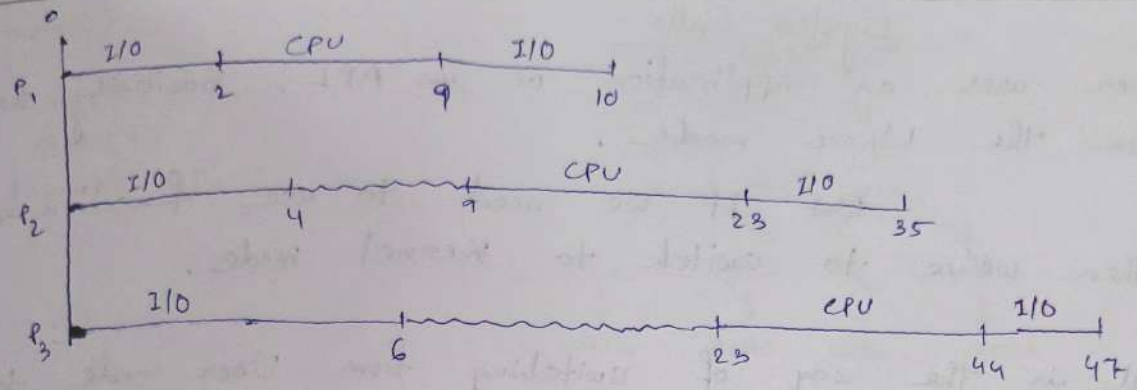
If a job isn't responding for a certain time, it's priority will be increased.

After executing for a certain time, if the job isn't completed the priority gets decreased.

Q Consider three processes, all arriving at time 0, with total execution time of 10, 20 and 30 units respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The OS uses a shortest remaining compute time first scheduling algo and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of cpu does the CPU remain idle?

Process	b.t	I/O burst	CPU burst	I/O burst
P_1	10	2	7	1
P_2	20	4	14	2
P_3	30	6	21	3

Shortest Remaining Time First



OS (L)

Switch Case :

```
case $ v. name in
    value 1) operation
        ;;
    value 2) operation
        ;;
    default : operand
    esac
```

Loop :

```
while [ condn ]
do
:
done
```

```
i=0
while [ $i < 10 ]
do
    echo $i
    i='expr $i + 1'
done
```

prints from 0 to 9

Write a shell script to print the entered character is a vowel or consonant. ~~or int~~

Write a shell script to print the day of a week using switch case.

Write a shell script to check whether entered no is prime or not.

to print n no of fibonacci series.

to find factorial of a given no.

to check entered no is palindrome or not (any length)

to find out prime nos present within a range.

System Calls

When a user uses an application or an API, basically the user is in the User mode.

But if we need to use functionality of OS then we've to switch to 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 that's why we need to switch to the Kernel mode to access via OS.

System Calls

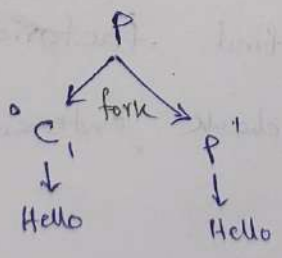
- File Related => Open(), Read(), Write(), Close(), Create file
- Device Related => Read, Write, Reposition, ioctl, fcntl
- Information => get Pid, attributes, get system time and date
- Process Control => Load, Execute, abort, fork, Wait, Signal, Allocate
- Communication => Pipe(), Create/delete connections, Shmget()

Fork :

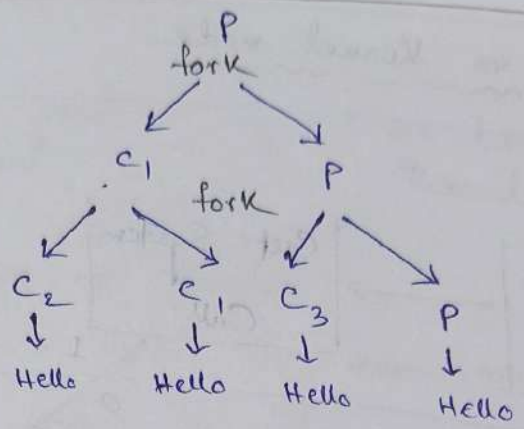
Suppose there is a program that can be referred as Parent program. If we use 'fork' inside the parent program it'll create a child program (basically clone of parent program).

Fork → 0 (child)
→ +1 (Parent)

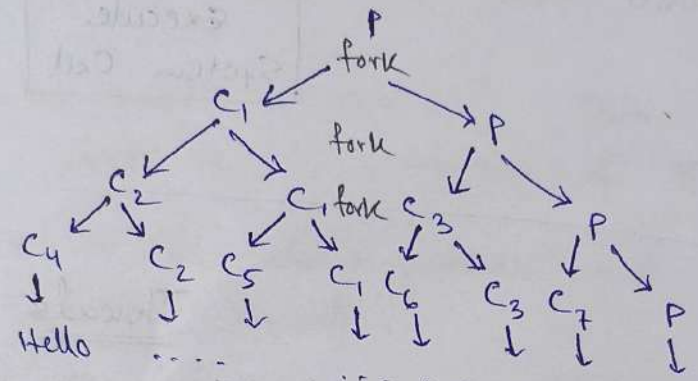
```
main()
{
  fork();
  printf("Hello");
}
```



```
main()
{
  fork();
  fork();
  printf("Hello");
}
```




```
main()
{
  fork();
  fork();
  fork();
  printf("Hello");
}
```

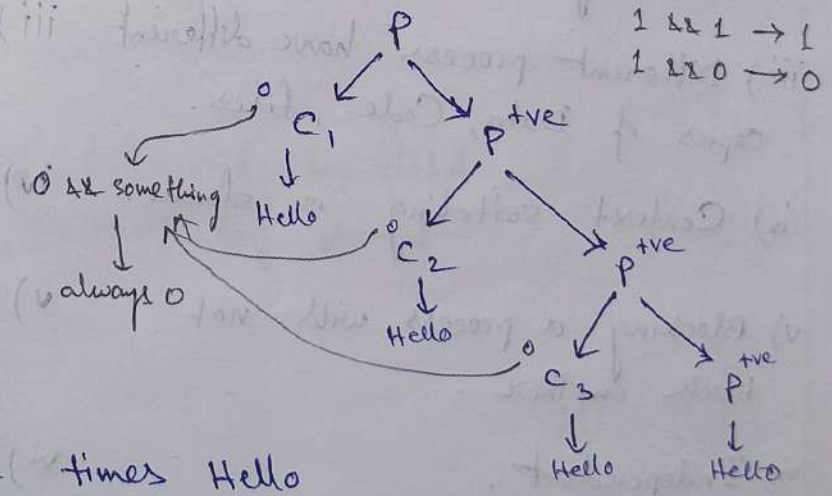


printf("Hello") will run → 2^n times → Total 8 times

where n = no of fork() used.

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

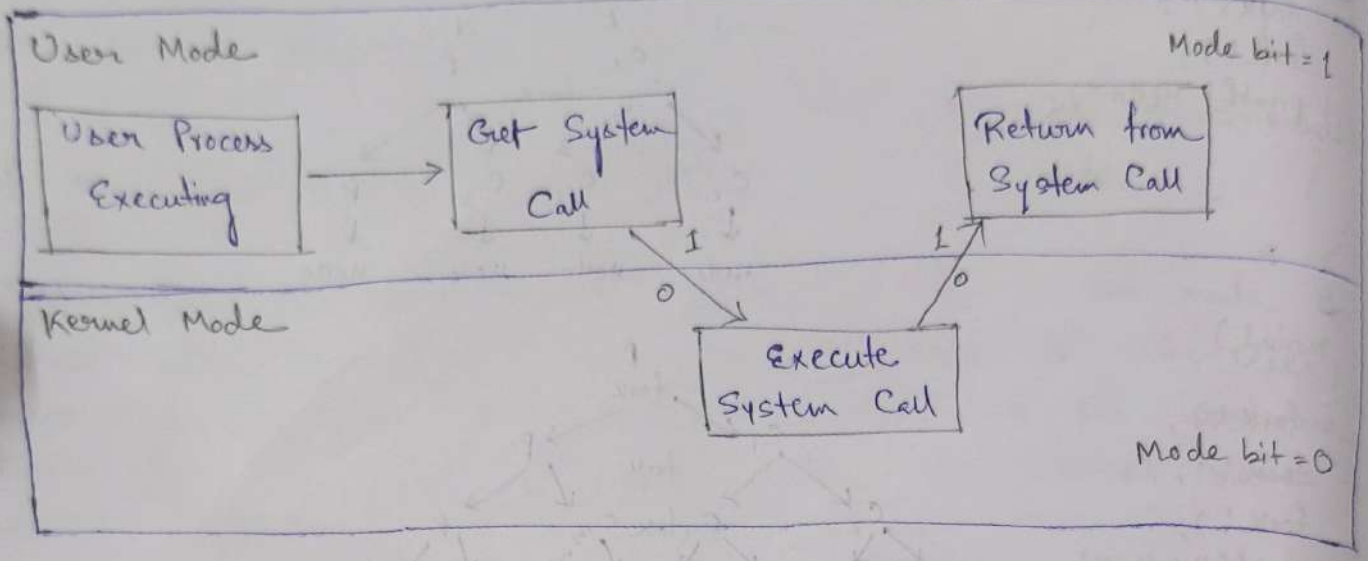
```
#include <stdio.h>
#include <unistd.h>
int main()
{
  if(fork() && fork())
    fork();
  printf("Hello");
  return 0;
}
```



1 && 1 → 1
1 && 0 → 0

No of times Hello will be printed = 4

User mode vs Kernel mode



Process	Threads
i) System calls involved in process.	i) There is no system call involved
ii) OS treats different processes 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 manage by OS.
ii) User level threads are typically fast.	ii) Kernel level threads are slow 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 block no effect on other kernels.

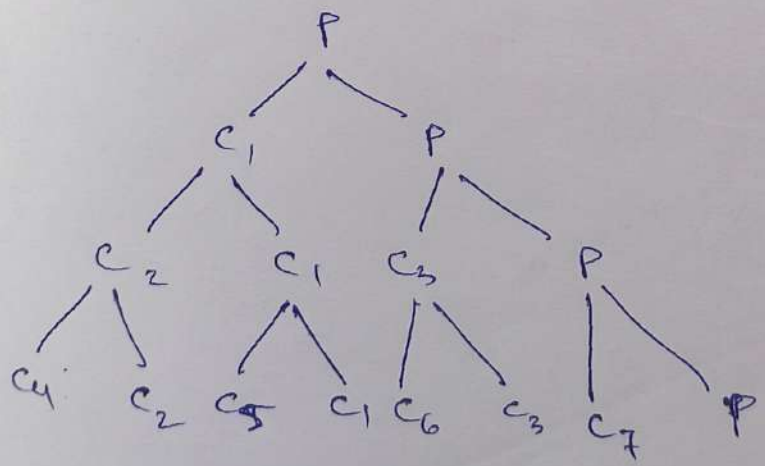
Context Switching Time

Process > Kernel Level Thread > User Level Thread

i) PCB holds information about a process.
ii) The exe code is loaded into the 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 zero 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();
```

Command interpretur is a program to create a process to run a program.

Starting address of the program to be executed.
Size of the memory allocated to the program
Content of PC
Registers
stack
Open files
Pointer pointing to opening of files.

Creation of a process:

Process mix

CPU bound process
I/O bound process

	CPU	I/O
P_1	60%	40%
P_2	90%	10%
P_3	40%	60%

P_1 and P_3 are optimally utilized by CPU & I/O processor

"But it also checks about the active state queue."

Process Synchronization

- Cooperative Processes
  - Shares
    - Variable
    - Memory
    - Code
    - Resource
      - CPU
      - Printer
      - Scanner
- Independent Processes

We'll only talk about cooperative or parallel processes where multiple processes runs in the same time. If the processes aren't properly synchronized they can create a problem (when they're sharing something). Suppose there are two processes P_1 and P_2 running parallely. Execution starts from P_1.

```
int Shared = 4

P1                          P2
X = Shared;                 Y = Shared;
X++;                        Y--;
Sleep(1);                   Sleep(1);
Shared = X;                 Shared = Y;
```

① ② ③ ④ ⑤

```
X = 5
X++ // X = 6                Y = 5
                            Y-- // Y = 4
Shared = 6                  Shared = 4
```

From P_1 and P_2 we can say at first we incremented it by 1 and then we decremented. So the value of Shared variable should be 5+1 = 6 & then 6-1 = 5 (as like initial). But here we see the final value of Shared = 4. This proves us that both P_1 and P_2 aren't synchronized properly & as they're sharing the same variable, incorrect result is coming. This is called Race Condition.

- **Producer Consumer Problem :**

Case-1

```
void consumer (void)
{
    int itemC;
    while (true)
    {
        while (count == 0);
        itemC = Buffer (out);
        out = (out+1) mod n;
        count = count - 1;
        process_item (itemC);
    }
}
```

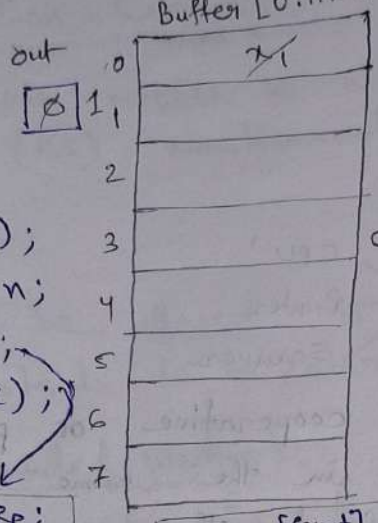
Buffer empty

```
int count = 0
void Producer (void)
{
    int itemp;
    while (true)
    {
        produce_item (itemp);
        while (count == n);
        Buffer[in] = itemp;
        in = (in+1) mod n;
        count = count + 1;
    }
}
```

Buffer full

n = 8

Buffer [0 .... n-1]




```
LOAD m[count], Rp;
DECR Rp;
STORE Rp, m[count];
```

```
LOAD m[count], Rp;
INCR Rp;
STORE Rp, m[count];
```

Suppose there is a producer who produces products and there is a consumer who consumes products. The execution starts from

i) Item x_1 is produced in produce_item func
ii) Checks whether the buffer is full → goes to infinite loop if full
iii) Stores the produced item in nth index of buffer. Initially in = 0.
iv) Value of 'in' is incremented to store next produced item. in = (0+1) % 8 = 1 % 8 = 1
v) count variable determines how many products are there in the buffer. Count will be incremented from 0 to 1.

---

i) Then control suppose comes to consumer.
ii) Checks whether buffer is empty (if empty goes to infinite loop). If the buffer is empty there's nothing to consume.
iii) Takes out x_1 (previously produced) and stores in 'itemC' variable.
iv) out will be incremented from 0 to 1. If further consuming occures it'll refer to 1st index. out = (0+1) % 8 = 1
v) count value will be decremented as there's no product in buffer now.
vi) process_item func will consume the 'x_1'.

Here, count variable and buffer are used by both the processes. This is the best case where everything works perfectly fine.

Case - II

```
void consumer (void)
{
  int itemC;
  while (true)
  {
    while (count == 0);
    itemC = Buffer [out];
    out = (out + 1) mod n;
    count = count - 1;
    process - item (itemC);
  }
}
```

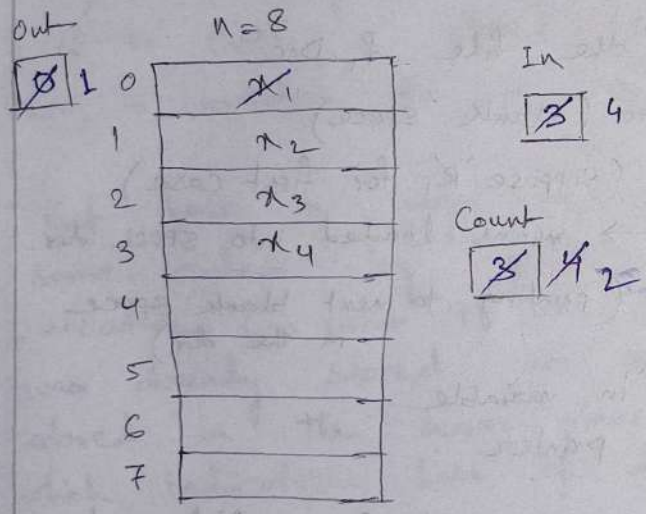
```
int count = 0
void producer (void)
{
  int itemp;
  while (true)
  {
    produce - item (itemp);
    while (count == n);
    Buffer [in] = item;
    in = (in+1) mod n;
    count = count + 1;
  }
}
```

1. LOAD m[count], R_p;
2. DECR R_p;
3. STORE R_p, m[count];

1. LOAD m[count], R_p;
2. INCR R_p;
3. STORE R_p, m[Count];

I_3 ↑ Consumer

Flow → Producer → Count++ → I_1, I_2 → Consumer → Count-- → I_1, I_2 → Producer → I_3



Suppose three products are already produced. So currently count = 3. Now another product x_u is being produced.

I) x_u is inserted into buffer
II) Previously In = 3 (pointing)
III) Now In = 3+1 = 4 (next pointing)
IV) Inside count = count + 1
↳ Suppose first two instn. execute then context switching occurs before storing the updated value of count from 3 to 4. So count remains 3.
V) Control goes to consumer.
VI) x_1 is taken out to consume.
VII) out is incremented to 1 (pointing to next product)
VIII) In the count = count - 1
↳ Executing 2 instn. context switching occurs before storing the updated value of Count 3 to 2. So count remains 3.

IX) Now control goes to producer & the 3rd remaining instn. encounters (R_p = 4). Now Count = 4

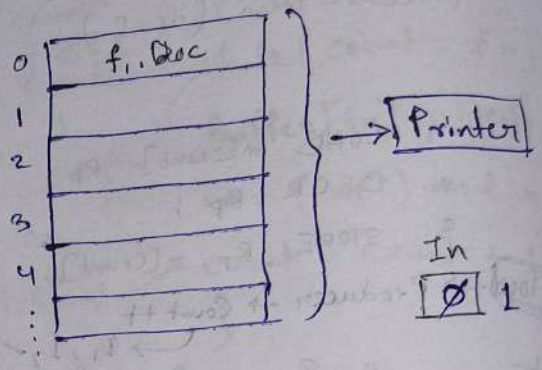
X) Then control goes to consumer to execute the 3rd instn. of count-- (R_p = 2). Now Count = 2.

Now in buffer there're currently 3 products but count = 2 which is faulty. We produced 1 product & consumed it. So count = 3+1 = 4 & then count = 4-1 = 3 again. So in this case it's not synchronized properly.

Printer Spooler Problem

Suppose there's only a single printer and multiple users are trying to access it. So there's a program called Spooler which takes multiple files and from the spooler all the files are sent to the printer one by one. But all of the files must be stored inside the Spooler directory before printing.

```
LOAD   Ri , m[in]
STORE  SD[Ri] , "F-N"   → filename
INCR   Ri
STORE  m[in] , Ri
```

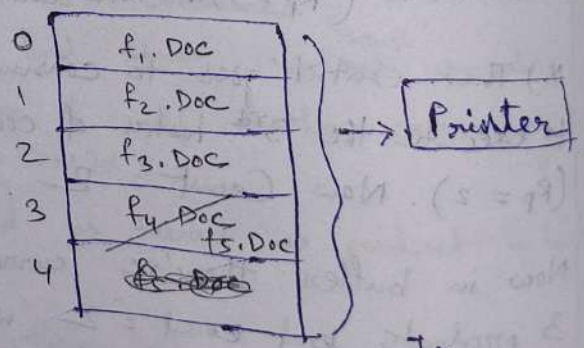


Best Case

i) P_1 process comes that contains the file $f_1.Doc$
ii) Initially 'in' is pointing to zero (blank space)
iii) value of 'in' stored into R_i (Suppose 'R' for first case)
iv) $f_1.Doc$ will be stored to R_1 → means loaded into SPOOL dir.
v) R_1 is incremented from 0 to 1 (pointing to next blank space in the dir)
vi) Storing the value of 'R' into 'in' variable.
vii) Then $f_1.Doc$ will be sent to printer.

Case II Now suppose there's two processes P_1 and P_2 which contains $f_4.Doc$ and $f_5.Doc$. And in the spool directory there's already 3 ~~predoc~~ files. Execution starts from P_1

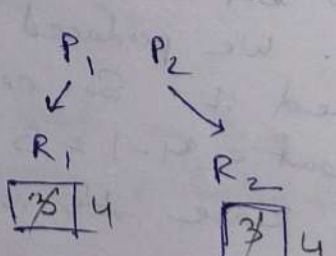
```
I1   LOAD   Ri , m[in]
I2   STORE  SD[Ri] , "F-N"
I3   INCR   Ri
I4   STORE  m[in] , Ri
```



Flow →

$P_1(I_1\ I_{11}\ I_{111}) \rightarrow P_2(I_1 I_{11} I_{111} I_{1V}) \rightarrow P_1(I_{1V})$

Preemt Preemt



i) P_1 process starts.
ii) Initially in = 3 (blank space as there's already 3 files)
iii) P_1 suppose has R_1 GPR so 3 is stored into R_1
iv) f_4.Doc is stored into R_1 (spool dir at in = 3)
v) R_1 is incremented from 3 to 4
vi) Context switching occurs before updating the 'in' value. So 'in' value remains as 3 like before.
vii) Now control goes to P_2
viii) P_2 has GPR R_2 and in = 3 so 3 is stored into R_2
ix) f_5.Doc is stored into R_2 (spool dir at in = 3)
x) R_2 is incremented from 3 to 4
xi) in value is updated from 3 to 4 (value of R_2 moved to 'in')
xii) Now P_2 is completed & control goes back to P_1
xiii) P_1 has only one instru. left that is updating the value of 'in' that is stored inside R_1
xiv) 'in' updates to 4 (as R_1 stores 4)

But here a major problem occurs. Due to sharing the same codes by multiple processes and incorrect synchro-nization (in point 9) we can see that previously f_4.Doc was already stored into the in = 3 but now f_5.Doc is stored in the same place (in = 3). So f_4.Doc is now gone which indicates loss of data.

Running State → Process Ideal State → Program

- **Critical Section Problem**

It is a part of the program where the shared resources are accessed by the various processes.

Critical section is a place where shared variables, resources are placed. (Common part b/w two different programs.)

Every program will have to go through Entry section before entering critical section and Exit section after leaving critical section. By this we can achieve Synchronization & can avoid Race Condition.

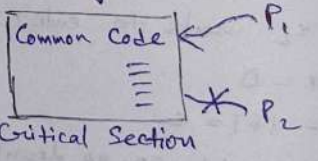
To achieve Synchronization, 4 conditions must be followed:

Primary rules { I) Mutual Exclusion
II) Progress

Secondary rules { III) Bounded wait
IV) No assumption related to hardware & speed.

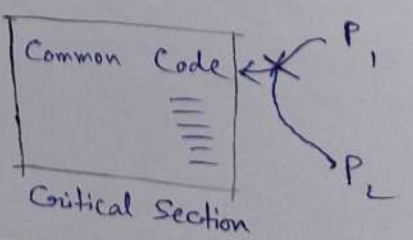
Mutual Exclusion:

If there are two processes P_1 and P_2. Suppose P_1 is currently using critical section then P_2 is not allowed to use the critical section. To achieve this we'll write the code for entry section of P_1 in such a way that when P_1 is using this then P_2 can't enter.



Progress:

Suppose in the critical section currently there's no process using it. P_1 is more interested than P_2 to use it. P_1 is trying to enter into the critical section but the other process P_2 is trying to prevent P_1. That means P_2 is not entering as well as blocking the progress of P_1. This should be avoided.



Bounded Wait :

Suppose P_1 is accessing the critical section and after completio it left. After a while P_1 enters again. Thus P_1 enters into critical section again and again while P_2 is still waiting for it's first turn to use the critical section. So the waiting time for P_2 should not be infinite. Staravation should not occur. Waiting time should be less.

No assumption related to hardware & speed :

Processes should be compatible with any type of architechture (32 or 64 bit) or any speed processor (1GHz or 7GHz) or any OS (Linux or Windows) etc. A solution should be universal.

Critical Section Solution Using "Lock"

```
do {
    acquire lock
        critical section (cs)
    release lock
}
```

```
1. while (lock == 1) ;      → ENTRY CODE
2. lock = 1 ;
3. Critical Section
4. lock = 0 ;               → EXIT CODE
```

lock = 0 → Vacant
1 → Full

*) Executes in User mode (Using API)
*) Multiprocess solution.
*) No mutual exclusion guvranty.

Best Case

1) Suppose P_1 and P_2 two processes are there.
2) P_1 tries to enter (initially lock = 0)
3) While encountering 2nd line lock = 1
4) P_1 enters into the critical section
5) In this stage suppose P_2 wants to enter (Now lock = 1 already).
6) In the 1st line for P_2, it goes to infinite loop as lock == 1.
7) Until P_1 ends the cs section & exits from it, then value of lock = 0 means gate is again open for another process.
8) Now P_2 can enter and again the lock = 1.

1. while (lock == 1);
2. lock = 1;
3. critical section;
4. lock = 0;

Case - II

1) Suppose P_1 wants to enter.
2) Encounters first line.
3) While going to second line, before setting the lock value from 0 to 1, P_1 got preemt.
4) Now P_2 wants to enter and P_2 successfully passed first line and set the lock value from 0 to 1.
5) P_2 enters into the critical section.
6) In this moment P_1 suddenly comes and starts from where it got preemt (2nd line). So P_1 set the lock value from 1 (set by P_2) to again 1.
7) Now P_1 also enters into the critical section.

So, mutual exclusion ~~fully~~ literally vanished and currently there are two processes in the critical section which is wrong.

- Critical Section Solution Using "Test_and_set" Instruction:

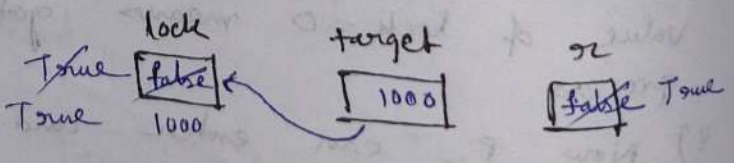
Previously we saw mutual exclusion vanishes if preemtion occurs b/w these two instructions.

```
while (lock == 1);
lock = 1;
critical section
lock = 0;
```

So in "test_and_set" we merged the first two instructions into a single instruction to achieve mutual exclusion.

```
while (test_and_set (& lock));
critical section;
lock = false;
```

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



Turn Variable (Strict Alteration) :

*) Applicable to 2 programs only.

*) Runs in user mode.

Suppose there are two processes P_0 and P_1

int turn =

P_0	P_1	
Non CS ≡	Non CS ≡	
while (turn != 0);	while (turn != 1);	→ ENTRY Code
critical section;	critical section;	
turn = 1;	turn = 0;	→ EXIT Code

1) Suppose initially turn = 0 so P_0 will enter because P_1 will stuck in the while loop (0 != 1);

2) P_0 enters into the critical section.

3) While exiting from CS; P_0 sets the turn value to 1.

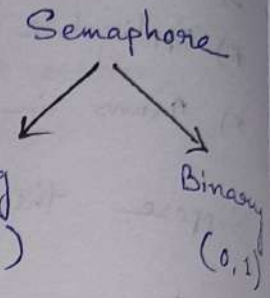
4) Now P_1 can enter after exiting from while loop (1 != 1).

So, 'Strict Alteration' definitely assure Mutual Exclusion But when it comes to progress it doesn't satisfy bcz when turn = 0 if P_1 wants to enter into CS, it can't. P_1 will have to wait until P_0 finishes it's CS and sets the turn = 1. So P_0 is blocking P_1 (kind of :)).

In case of Bounded Wait, strict alteration provides it bcz each & every time either P_1 or P_0 is entering into CS alternatively. So starvation isn't happening.

Semaphore :

Semaphore is an integer variable which is used in mutual exclusive manner by various concurrent cooperative processes in order to achieve Synchronization.

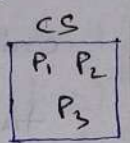


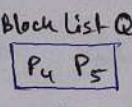
Entry Code : P(), Down, Wait.

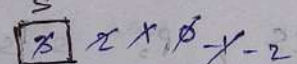
Exit Code : V(), Up, Signal, Post, Release.

Entry Section

```
Down (Semaphore S)
{
    S.value = S.value - 1;
    if(S.value < 0)
    {
        Put Process (PCB) in
        suspended list, sleep()
    }
    else
        return;
}
```







Suppose S = 3

1) P_1 wants to enter into CS
   S = 3 - 1 = 2
   it doesn't enters into the 'if' and directly enters CS

2) P_2 wants to enter
   S = 2 - 1 = 1
   it also enters

3) Likewise P_3 enters
   S = 1 - 1 = 0
   if(0<0) → false → enters CS

4) In case of P_4
   S = 0 - 1 = -1
   it enters into 'if' block and temporarily gets blocked
   ↓
   block list queue.

5) Same goes for P_5
   directly goes to block list queue

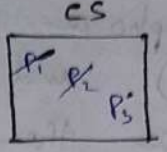
From this we can easily say that S = -4 means in the block list queue there're already 4 processes.

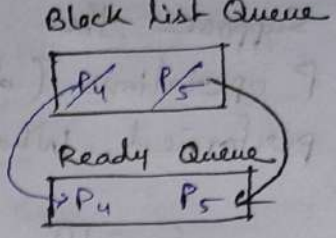
S = 0 means it'll be the last process that'll enter into CS. After that each process will go to block list queue.

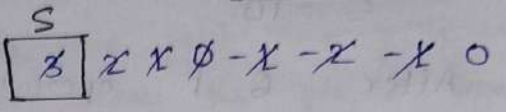
S = 10 means 10 process can successfully enter into CS.
↳ 10 successful operations can be performed.

Exit section

```
Up ( Semaphore S)
{
    S.value = S.value + 1 ;
    if (S.value ≤ 0)
    {
        select a process from
        suspended list, wake up();
    }
}
```







1) Suppose P_1 complete it's work in CS and now want to exit.
2) But before exiting it'll have to go through the exit section code.
3) Currently $S = -2$ when P_1 is exiting S.value = $-2+1 = -1$
4) $(-1 \leq 0)$ → True → enters into 'if' block.
5) Now following the FIFO order P_4 is taken out from the block list queue & moved to the ready/active state queue.

Now P_2 wants to leave.

1) $S = -1 + 1 = 0$
2) $(0 \leq 0)$ → True → enters into 'if' block.
3) Now P_5 will be moved to ready queue.

Case-I Now $S = 0$,

if P_4 wants to enter from the ready state queue! (into CS)

1) it'll run down code first.
2) $S = 0 - 1 = -1$
3) Goes to 'if' block as $(-1 < 0)$ ↓ True
4) Again goes to block list queue
5) So can't enter until P_3 leaves

Case-II Now $S = 0$,

P_3 wants to leave.

1) $S = 0 + 1 = 1$
2) Doesn't go to 'if' as $(1 > 0)$
3) Simply exits from Up.
4) After the departure of P_3 now P_4 can try to enter into CS.

Q) Suppose the Semaphore value is 10. At first 6 successful P operations (down) and 4 successful V operations (O_P) are performed. What is the current value of S (semaphore)?

Ans)

S = 10

After 6 P operations value of S = 10 - 6 = 4.

After 4 V operations value of S = 4 + 4 = 8.

Q) S = 17, 5 P operations & 3 V operations & 1 P operation in sequence. Value of S = ?

Ans)

After 5P operations S = 17 - 5 = 12

" 3V " S = 12 + 3 = 15

" 1P " S = 15 - 1 = 14

Binary Semaphore

It is easy to use because the value can be only 0 or 1.

```
Down (Semaphore S)
{
if (S.value == 1)
{
    S.value = 0;
}
else
{
    Block this process
    and place in suspend
    list, sleep();
}
}
```

Entry section

1) If Semaphore value is 1 while entering into 'Down' code it'll enter into the 'if' block and set the value of s.value from 1 to 0
↳ "Successful Operation"
The process then can easily enter into the CS.

11) If the Semaphore value is 0 while entering into 'Down' code the process will be sent to block list queue.
↳ "Unsuccessfull Operation"

```
Up(Semaphore S)
{
    if (Suspend list is empty)
    {
        S.value = 1
    }
    else
    {
        Select a process from
        suspend list and wake up();
    }
}
```

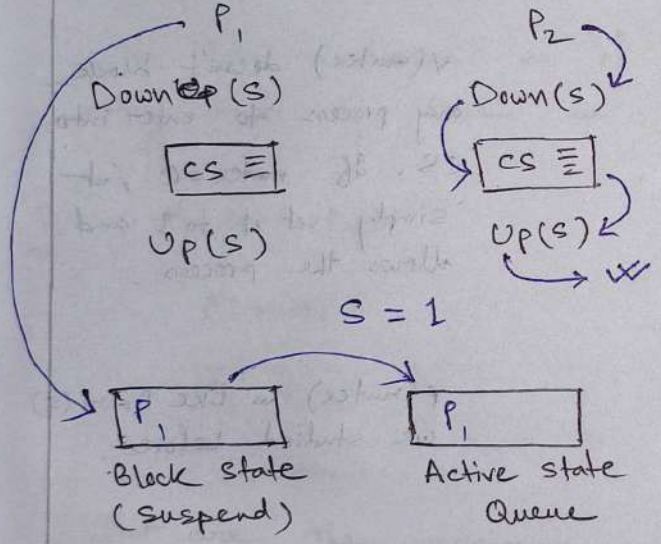
Exit section

1) At first it'll check if there're any processes in the block list queue / suspend state, if any process is there it'll jump to the else part (keeping the s.value=0)

or 1) and move the process from block list queue / suspend state to active / ready state queue.

ii) If there're no processes in the suspend state, then it'll simply update the s.value from 0 to 1 and will allow new processes to come into CS.

Suppose there're two processes P_1 and P_2.



i) Suppose any of the processes want to enter into CS, but if we set S=0 both the processes will get blocked.

ii) let's say S=1 initially and P_2 wants to enter.

iii) So P_2 encounters down, goes into 'if' block and makes S=0.

iv) Now P_2 is inside CS.

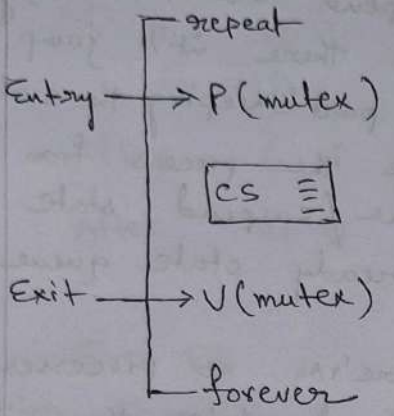
v) If P_1 now tries to enter into CS it'll get blocked. bcz when it'll encounter down (S=0) it'll go to else part.

vi) After completion in CS, now P_2 wants to leave. Before that it'll have to encounter 'Up' code.

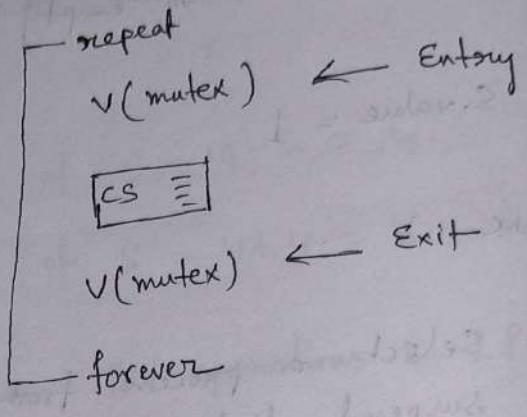
vii) P_1 is in the block state so at first it'll be moved to active state queue so that P_1 can enter into CS later.

viii) P_2 leaves.

Q) Each process P_i {i=1 to 9} executes the following code

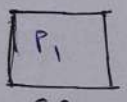


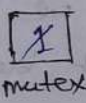
Process P_{10} executes the following code



What is the maximum no of processes that may present in CS at any point of time?

Ans ①





i) Suppose P_1 wants to enter

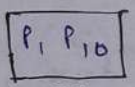
ii) Encounters P(mutex), set the value from 1 to 0

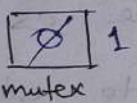
iii) Now, $P_2, P_3, \ldots P_9$ any process that want to enter will be sent to block state.

V(mutex) doesn't block any process to enter into CS. If mutex = 0, it simply set it to 1 and allows the process.

P(mutex) is like Down(s) we studied before.

②

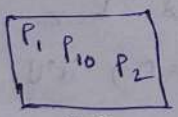


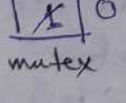


i) But P_{10} is using diffn algo for entry section.

ii) Using V(mutex) P_{10} enters into CS and now mutex = 1

③



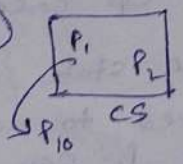


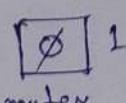
i) As mutex = 1, $P_2, \ldots P_9$ can try to enter.

ii) Suppose P_2 tries and successfully enters into CS and now mutex is again set to 0

V(mutex) always sets the mutex value to 1 without checking any condition.

④



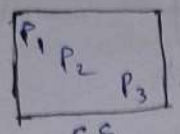


i) At this point no new process can enter.

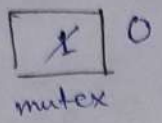
ii) Suppose P_{10} wants to leave.

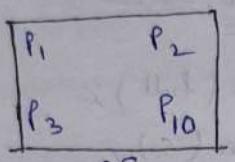
iii) Encounters V(mutex) in the exit section and sets mutex from 0 to 1

iv) Now any one can enter.

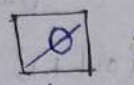
(5) 

i) Now P_3 wants to enter
ii) Encounters P(mutex) and as mutex = 1, after entering CS mutex becomes 0 again.



(6) 

i) Suppose P_{10} again wants to enter
ii) Using V(mutex), sets mutex from 0 to 1
iii) Now 4 processes are in CS
iv) $P_4 \dots P_9$ anyone can also enter again



⋮

Thus P_{10} will enter and leave several times and will create a scope other processes to enter into the CS. There'll be a stage where all the 10 processes (P_1 to P_{10}) will be in CS.

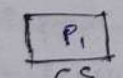
∴ The maximum no of processes that may present in CS at any point of time is 10.

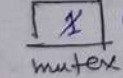
Q

P_1 to P_9	P_{10}
P(mutex)	V(mutex)
CS ≡	CS ≡
V(mutex)	P(mutex)

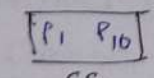
What are the maximum no of processes that may present in CS at any point of time?

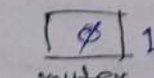
Ans

(1) 

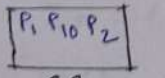


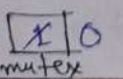
i) P_1 wants to enter
ii) Using P(mutex) sets mutex to 0.
iii) Now $P_2 \dots P_{10}$ can't enter.

(11) 

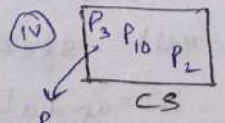


i) P_{10} enters using V(mutex)
ii) Sets mutex to 1
iii) P_1 and P_{10} are inside CS

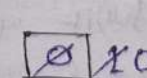
(111) 



i) P_2 enters using P(mutex)
ii) $P_3 \dots P_9$ can't enter as mutex = 0
iii) Now if P_{10} leaves mutex will be still 0 (not 1) bcz this time P_{10} is using P(mutex) for exit section

(IV) 

i) If P_1 leaves again mutex becomes 1
ii) $P_3 \dots P_9$ anyone can enter.



iii) P_3 enters and mutex = 0

∴ The maximum no of processes that may present in CS at any point of time is 3.

- **Solution of Producer - Consumer Problem Using Semaphore**

Counting Semaphore → full = 0 = No of filled slots.
→ empty = N = No of empty slots.

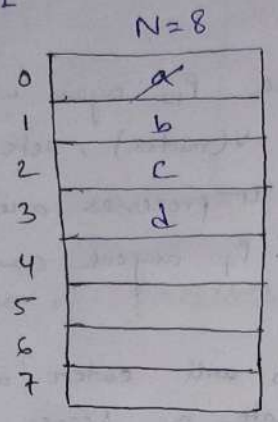
Binary Semaphore, S = 1

Case-I Best Case

Produce_item (item P)
=
down (empty);
down (S);

CS
```
Buffer[in] = itemp ;
In = (In+1) mod n;
```

up(S);
up(full);



In: 3 4

Consumer
=
down (full);
down (S);

CS
```
item C = Buffer[out];
out = (out+1) mod n;
```

up(S);
up(empty);

Out: 0 1

1) Suppose there're already 3 products a, b, c produced by the producer.
11) Now In = 3
111) Producer wants to make another product that is 'd'.
1v) Now empty = 5, full = 3 and S = 1 (initially)
v) Encounters down (empty) makes it 4 (from 5)
v1) " down (s) makes it 0 (from 1)
v11) Enters into cs.
v111) 'd' is successfully loaded into buffer at 3rd index.
1x) In = (3+1) % 8 = 4 % 8 = 4 updated in value
x) Encounters exit section.
x1) In up(S) it updates S to 1 (from 0)
x11) In down(full) it updates full to 4 (from 3)
x111) Producer successfully exits.

Now consumer wants to consume a product.
1) Encounters down (full) makes it 3 (from 4)
11) " down (S) " " 0 (from 1)
111) Enters into cs.
1v) Out is pointing to 0 that is 'a' product
v) Takes the 'a' product out of buffer.

vi) out = (0+1) % 8 = 1 % 8 = 1 pointing to next product to consume.

vii) Encounters Up(s) and makes it 1 (from 0)

viii) Encounters Up(empty) and makes it 5 (from 4)

Empty = ~~5~~ 4 5

full = ~~2~~ 4 3

S = ~~1~~ ~~0~~ ~~1~~ ~~0~~ 1

Case-II (In case of pre-emtion)

Produce - item (itemP)

=

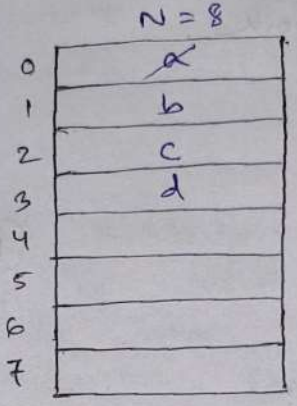
down (empty);

down (s);

CS
```
Buffer[in] = itemp;
In = (In+1) mod n;
```

up(s);

up(full);



· Consumer

=

down(full);

down(s);

CS
```
itemC = Buffer[out]
out = (out+1) mod n;
```

up(s);

up(empty);

~~3~~ 4 In

~~0~~ 1 Out

empty = ~~5~~ 4 5 full = ~~3~~ 2 3 S = ~~1~~ ~~0~~ ~~1~~ ~~0~~ 1

i) Suppose there are 3 products a, b, c produced by the producer.

ii) In = 3

iii) Starts with ~~consumer~~ producer

iii) down(empty) → empty updated to 4 (from 5)

iv) But before going to down(s), context switching occurs.

v) down(full) → updated to 2 (from 3)

vi) down(s) → updated to 0 (from 1)

vii) 'a' is taken out.

viii) out = (0+1) % 8 = 1

ix) Suppose now again context switching occurs.

x) Producer now encounter down(s) but s = 0, if producer wants to go further it'll get blocked. (That's the solution)

xi) So back to consumer and encounters up(s) → s becomes now 1.

xii) up(empty) → empty becomes 5 (from 4)

xiii) Consumer is done & control comes back to producer.

xiv) Producer's down(s) → s becomes 0 (from 1)

xv) new product 'd' is added to 3rd index in buffer.

xvi) In = (3+1) % 8 = 4 % 8 = 4 (new pointer)

xvii) up(s) → s becomes 1 (from 0)

xviii) up(full) → full updates to 3 (from 2)

Reader - Writer Problem (Solution Using Semaphore)

There are two types of users that are accessing a database. One is reader and the other is writer.

While accessing the same data,

- R - R ⟶ No Problem
- R - W ⟶ Problem
- W - R ⟶ Problem
- W - W ⟶ Problem

(This 3 should be removed)

while accessing diff. data at the same time, there's no problem.

Problem can only be removed by achieving process synchronization. Here database is the control section.

rc = read count = no of readers in buffer.

```
int rc = 0
Semaphore mutex = 1
Semaphore db = 1
void Reader (void)
{
  while (true)
  {
    down(mutex);
    rc = rc+1;
    if(rc==1)
      down(db);
    up(mutex);

    DB ∈

    down(mutex)
    rc = rc-1;
    if (rc==0)
      up(db);
    up(mutex);
    process(data);
  }
}
```

Reader

```
void writer (void)
{
  while (true)
  {
    down(db);

    DB ∈

    up(db);
  }
}
```

Writer

Initial Values

mutex	db	rc
1	1	0

Case-I (R-W)

i) Reader enters
ii) mutex = 1, db = 1
iii) mutex = 0
iv) rc = 0 + 1 (No of reader increases)
v) rc = 1, so under if db = 0
vi) mutex = 1
vii) Reader R_1 is under database
viii) writer enters
ix) down db means decrement db from 1 to 0 but db is already 0 so writer gets blocked. That's what we want to avoid problem by granting reader - writer access on the same data at the same time.

RC	mutex	db
0 1	1 0 1	1 0

Case-IV (R-R)

i) Reader r_1 enters
ii) mutex = 1, db = 1, rc = 0
iii) rc = 0 + 1 = 1 & mutex = 0
iv) rc == 1 so, db = 0 (from 1)
v) mutex = 1
vi) r_1 is in the database
vii) Now r_2 wants to enter
viii) mutex = 0
ix) rc = 1 + 1 = 2
x) ~~down~~ rc ≠ 1 so no down(db)
xi) mutex = 1
xii) r_2 is also in the database
xiii) That's what we want! Multiple readers can access the same database at the same time.

rc	mutex	db
0 1 2	1 0 1	1 0

Case-II (W-R)

i) writer w_1 enters
ii) mutex = 1, db = 1, rc = 0
iii) db = 0 bcz of down(db)
iv) w_1 enters into database
v) Now reader r_1 wants to enter
vi) mutex = 0 bcz of down(mutex)
vii) rc = 0 + 1 = 1
viii) As (rc == 1) under if → db-- (change 1 to 0) but db is already 0 so reader r_1 gets blocked.
ix) That's what we want!

RC	mutex	db
0 1	1 0	1 0

x) reader is not allowed to access the database while writer is accessing the database.

Case-III (W-W)

i) writer w_1 enters.
ii) mutex = 1, db = 1, rc = 0
iii) db = 0 bcz of down(db)
iv) w_1 is now in the database.
v) Now another writer w_2 wants to enter.
vi) down(db) → should decrement db from 1 to 0 but db is already 0 so writer w_2 will get blocked.
vii) That's what we want.

RC	mutex	db
0	1	1 0

If a reader wants to leave the database

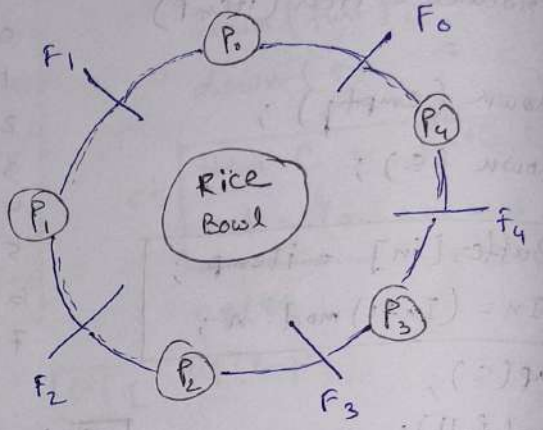
i) Suppose r_1 wants leave (consider after r-r)
ii) mutex = 0 (1 to 0)
iii) rc = 2 - 1 = 1
iv) rc ≠ 0 so no up(db)
v) mutex = 1 (0 to 1)
vi) r_1 is out of the code!
vii) Suppose now r_2 is in the database & writer w_1 can enter, w_1 will get blocked
viii) As soon w_1 encounters down(db) as db ~~is always 0~~ is already 0.
ix) So you can then bring out r_2 like r_1 before, rc = 0 (after r_2 leaves)

rc	mutex	db
1 0	1 0 1	0 1

Dining Philosophers Problem :

There are 5 philosophers eating in a table. Each of them having a fork. Philosopher has two states. One is thinking and the other one is eating.

While eating philosopher will take the left fork first and then the right fork and will start eating then. While eating is complete, will put the left fork first and then the right fork.




```
void philosopher (void)
{
  while (true)
  {
    Thinking ();
    take_fork (i);
    take_fork ((i+1) % N);
    EAT ();
    put_fork (i);
    put_fork ((i+1) % N);
  }
}
```

Case-1

i) Philosopher P_0 comes (willing)
ii) Thinking phase starts.
iii) " " ends.
iv) Takes the left fork which is F_0
v) " " right " " " $F_{(0+1) \bmod 5} = F_1$
vi) Starts eating.
vii) Ends eating.
viii) Put the left fork first F_0
ix) Put the right fork first F_1
x) Now P_1 comes.
xi) Thinking starts and ends.
xii) Takes the left fork which is F_1
xiii) " " right " " " $F_2 = F_{(1+1) \bmod 5} = F_{2 \bmod 5}$
xiv) Starts and ends eating
xv) Put the left fork first F_1
xvi) " " right " " F_2.

Normal Case

Case-II

i) Suppose P_1 comes.

ii) P_0 takes the left fork F_0

iii) But before taking the right fork F_1, suddenly P_1 enters and takes the left fork which is F_1

iv) Now P_0 will have to wait until P_1 finishes eating and puts back the left fork F_1.

This is a problem of starvation that must be removed. We use semaphore for this. We'll use that many no of binary semaphores that is same as the no of forks. We take array of semaphores. Philosophers with independent forks can eat at the same time. (Can access CS)

S[i] → Array of semaphores

S_0 S_1 S_2 S_3 S_4 (All are initialized with value 1)

```
void philosopher (void)
{
  while (true)
  {
    Thinking ();
    wait (take_fork (Si));
    wait (take_fork (S(i+1) mod N));
    EAT ();
    Signal (Put_fork (i));
    Signal (Put_fork ((i+1) % N));
  }
}
```

P_0 : S_0 S_1
P_1 : S_1 S_2
P_2 : S_2 S_3
P_3 : S_3 S_4
P_4 : S_4 S_0

S_0	S_1	S_2	S_3	S_4
↓	↓	↓	↓	↓
0	0	1	1	1

i) P_1 comes and takes the left fork F_0

ii) S_0 set to 0 (from 1)

iii) P_1 takes the right fork F_1

iv) S_1 set to 0 (from 1)

v) P_0 starts eating.

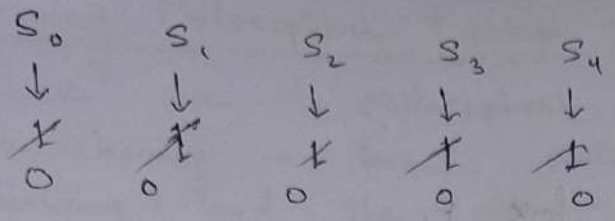
vi) If P_1 wants to come

vii) P_1 wants to take left fork F_1 but that Semaphore S_1 is already set to 0. So P_1 gets blocked.

viii) Now if P_2 wants to come & can, he/she can easily eat

ix) Because P_2 needs fork F_2 and F_3 which are not occupied by any of the philosopher and also the semaphore values of S_2 and S_3 are 1.

x) So likewise P_0, P_2 also takes fork F_2 and F_3 and starts eating with F_0 in CS. (Independent)



P_0 : S_0 S_1
P_1 : S_1 S_2
P_2 : S_2 S_3
P_3 : S_3 S_4
P_4 : S_4 S_0

i) Suppose P_0 comes.
ii) Takes the left fork F_0
iii) Gets preemted before taking right fork.
iv) P_1 comes and takes left fork F_1
v) Gets preemted before taking right fork.
vi) P_2 comes and takes left fork F_2
vii) Gets preemted before taking right fork.
viii) P_3 comes and takes left fork F_3.
ix) Gets preemted before taking right fork.
x) P_4 comes and takes left fork F_4.
~~xi) Gets preemted befor~~
xi) Now P_4 wants to take right fork F_0
xii) But semaphore value is 0 so P_4 gets blocked.

So in this case each of the philosopher can't start eating bcz each of them only have the left fork. This condition is called Deadlock.

This can be removed by adding a little modification in the code

i) For the first (n-1) philosophers they'll take left fork first and then the right fork.

ii) But for the n^{th} philosopher he/she will take the right fork first and then the left fork.