Syllabus

1. Basic Introduction : types, process diagram, system call
2. Process Scheduling : FIFO , SJF, Primitive, Round Robin
3. Process Synchronisation : semaphore
4 Deadlock and threads : Bankers algo
5. Memory Management : Paging, segmentation, fragmentation, virtual memory, page replacement algos
6. Disk Scheduling : SCAN, CSCAN, FCFS
7. Unix Commands : basic commands, open system call
8. File Management and Security : sequential, random, linked

Operating System and its Functions

→ To provide convinience to user

1. Resource Management :
   multiple users access same hardware
2. Process Management :
   execution of multiple processes
   CPU scheduling algos
   running program = process
3. Storage Management:
   how to store data
4. Memory Management : (RAM)
5. Security and Privacy

↓

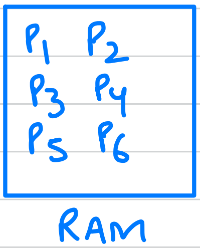
System Software (Works as an interface between user and hardware)

Types of Operating System

1. Batch (similar processes in one batch)
2 Multiprogrammed
3. Multitasking
4. Real Time OS
5. Distributed
6. Clustered
7. Embedded

Multiprogrammed

Non-preemptive : One process executes completely then next



Execution Order → $P_1, P_2, P_3, P_4, P_5, P_6$

The CPU executes one process completely then moves onto the next.

In between, if the process itself decides to pause (I/O), then only the other process is loaded to the CPU.

No idling of CPU in this case

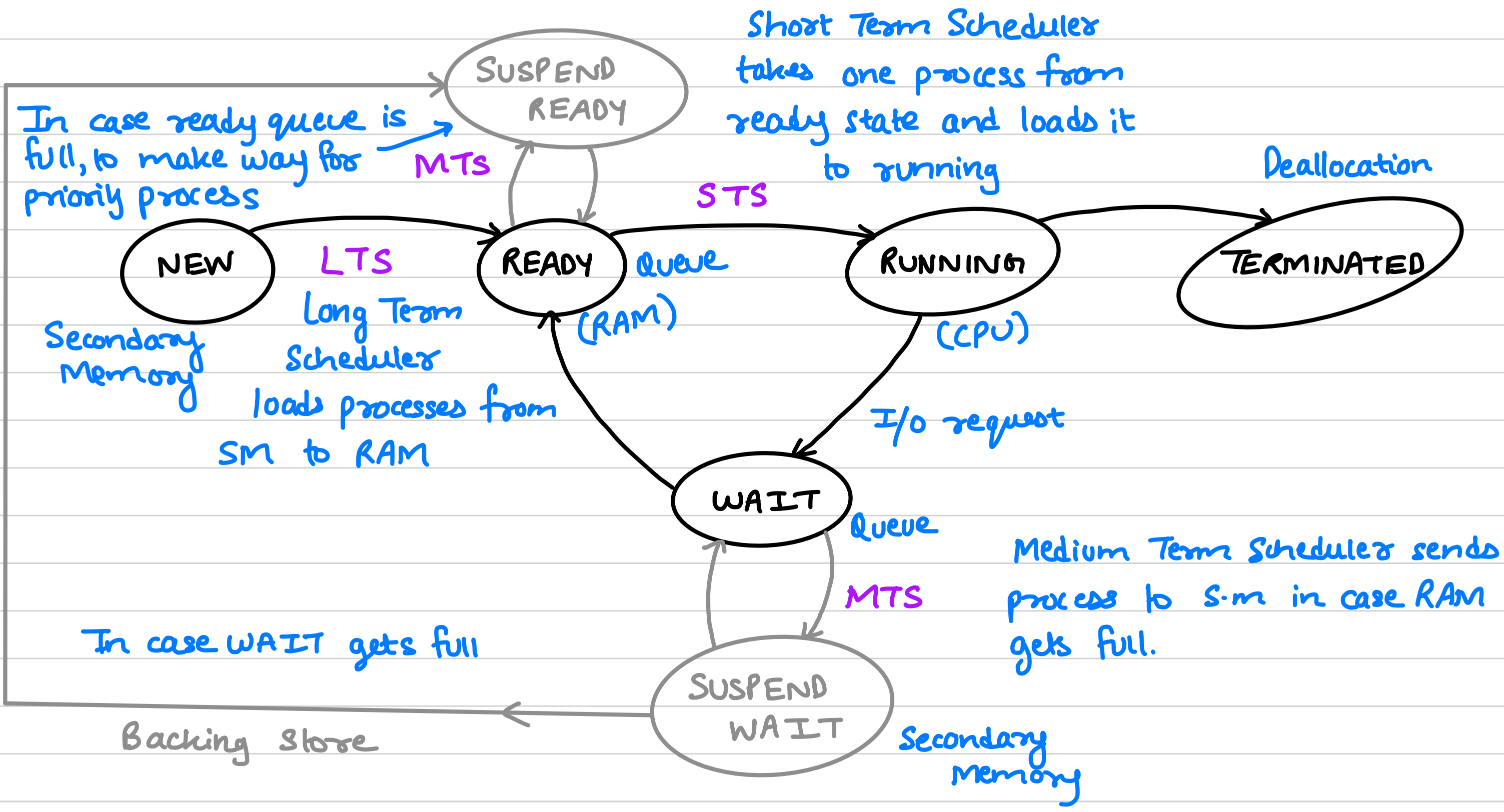
Multitasking / Time Sharing OS

Preemptive : A process is not executed completely and CPU moves to next

The CPU decides to pay attention to all programs. It executes first program and in between moves to another program. This increases responsivness.

Our computers have multitasking OS

Process State



Short Term Scheduler takes one process from ready state and loads it to running

In case ready queue is full, to make way for priority process

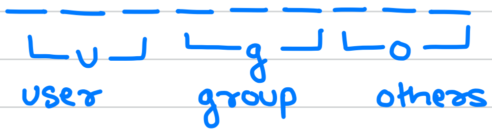
Long Term Scheduler loads processes from SM to RAM

Medium Term Scheduler sends process to S.M in case RAM gets full.

In case WAIT gets full

Important linux commands

chmod



read	r	4
write	w	2
execute	x	1

→ chmod ugo =r → read permission

↓ user, group, other

→ rx = 4+1 = 5

rw = 4+2 = 6

→ chmod 666 : read + write permission to ugo

System Call

When a process requests a service from kernel of the OS.

1. File related : open(), read(), write(), close()
2. Device related : read , write
3. Information : get pid (process id) , attributes
4. Process Control : load, execute , abort , *fork, wait , signal, allocate

↓ creates a child process and both work simultaneously making a multiprocessing environment

5. Communication · pipe() , create / delete connections
(Inter process communication)

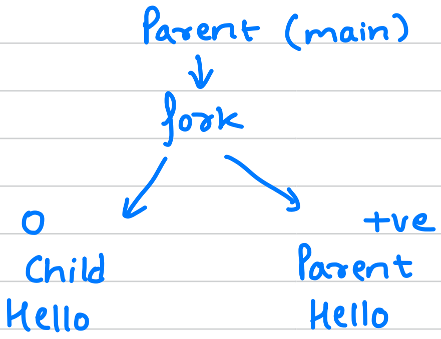
Fork() → 0 child

→ 1 +ve parent

→ -1 -ve (child process could not be created due to some reason)

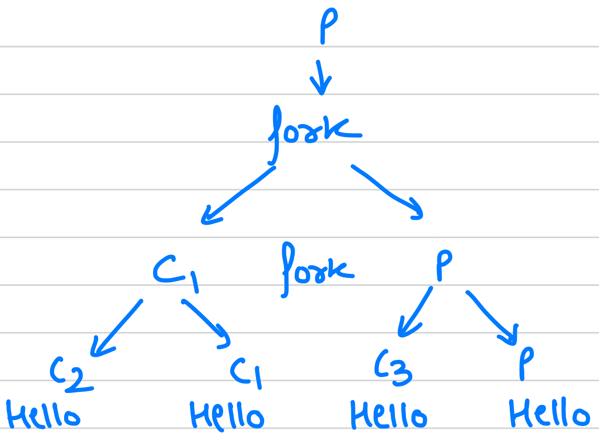
Eg:

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



Eg:

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



2^n

child process $= 2^n - 1$

Q

```
int main()
{
    int a;
    for (a=1; a<5; a++)
        fork();
    printf("1");
}
```

How many times "1" will be printed ?

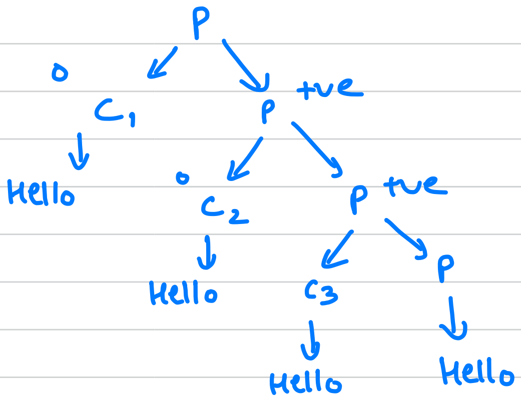
Ans $2^4 = 16$

Q

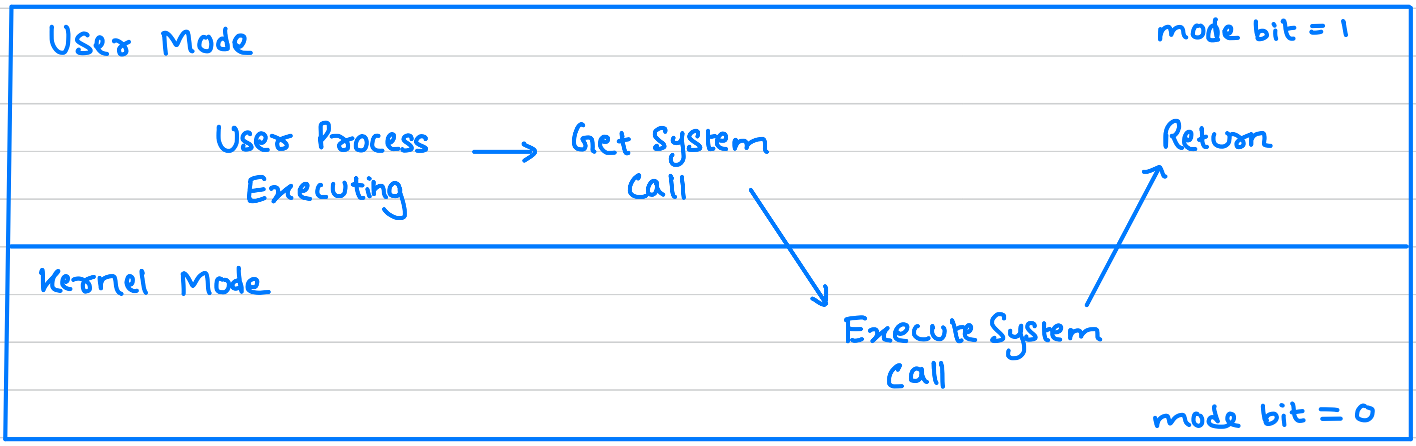
```
main() {
  if ( fork() && fork())
      fork();
  printf("Hello");
}
```

How many times "Hello" ?

Sol 4

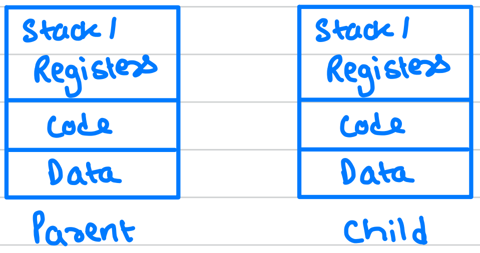
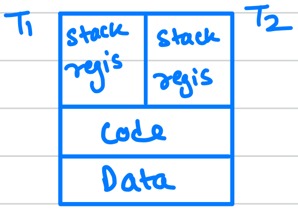


User mode and kernel mode



The CPU keeps switching between user mode and kernel mode.
Suppose the process demands I/O operation, the CPU will shift from user to kernel mode, execute a system call and then return.
(read())

Processes vs Threads

Process	Threads
 Stack / Registers Code Data Parent Stack / Registers Code Data Child	
→ Different processes have diff copies of data, files, code.	→ Threads share same copy of code and data
→ System calls involved in process. Child process is created through fork() system call.	→ No system calls involved as user is responsible for threading.
→ Parent and child process have diff pid	→ One pid
→ OS treats different processes differently	→ All user level threads treated as single task for OS
→ Context switching is slower	→ Context switching is faster
→ Blocking one process will not block another. (Independent)	→ Blocking a thread will block entire process (Interdependent)