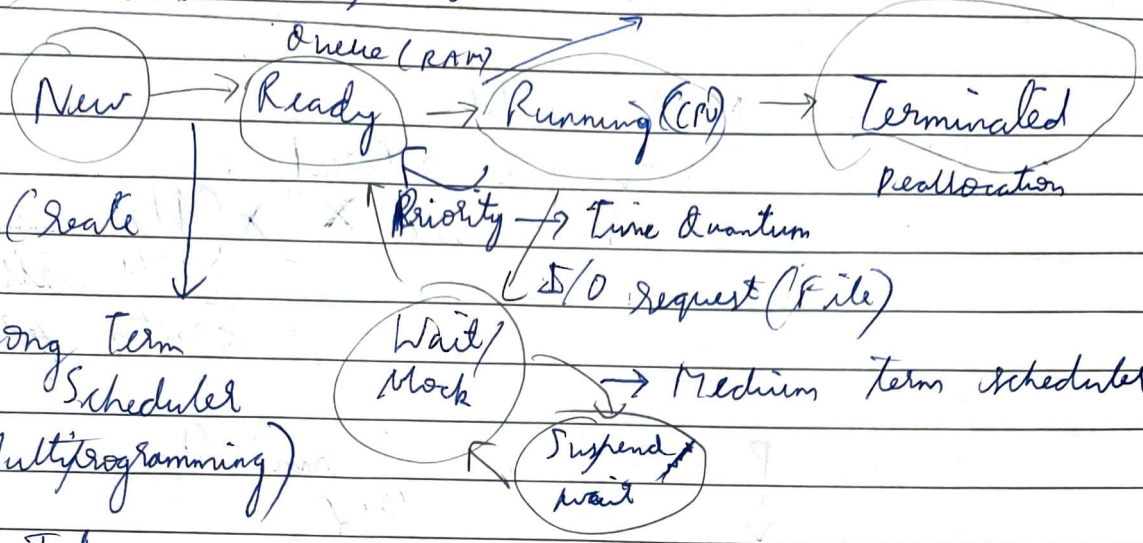
Short-term Scheduler

↑

Dispatched (Schedule) (RAM)

1.5 Process states in OS



If the CPU does not do multitasking (or completes ~~all processes~~ the process in time) it is called non-preemptive. If not, it is called preemptive.

If all the processes running in the CPU goes to wait state the processes are sent to suspend wait state that is an additional state. This particularly happens when the queue for the wait state is filled.

In this case, all the processes are swapped out to the second memory.

If in the queue in the RAM a kernel process (with high priority) occurs, the other processes are sent into an additional state called suspend ready by Medium Term Scheduler. Also, if the queue at suspend wait state is high, all the process are sent to suspend ready state. This method is called locking store

L-1.7 System Calls in Operating System & its Types

System Call

→ File related ⇒ Open(), Read(), Write(), Close(), Create file etc.

→ Device Related ⇒ Read, Write, Reposition, ioctl, fnctl

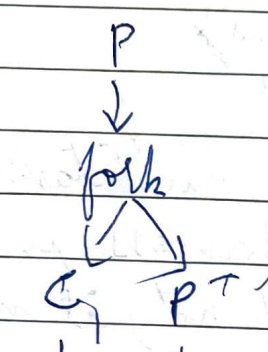
→ Information ⇒ get Pid, attributes, get System time and data

→ Process Control ⇒ Load, Execute, abort, Fork, Wait, Signal, Allocate etc.

→ Communication ⇒ Pipe(), Create/delete connections, Shmget()

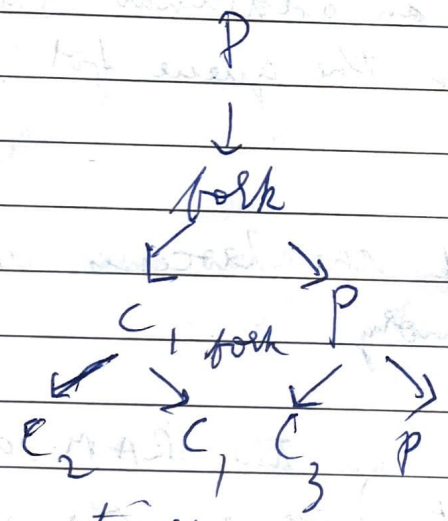
L-1.8 : Fork system call with Example

fork() → 0 → child
+ve → parent
-1 → child ✗] ✗ (assume that child will be created for sure)



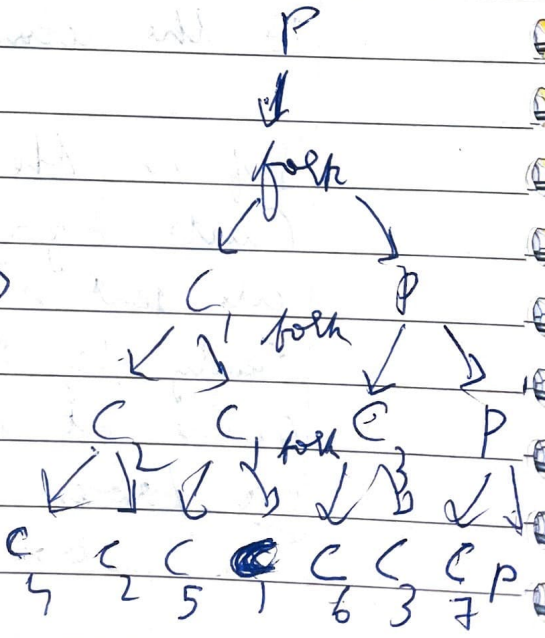

```
main()
{
  fork();
  printf("hello");
}
```

if 2_n fork() is written



times

if 3_n fork() is written →

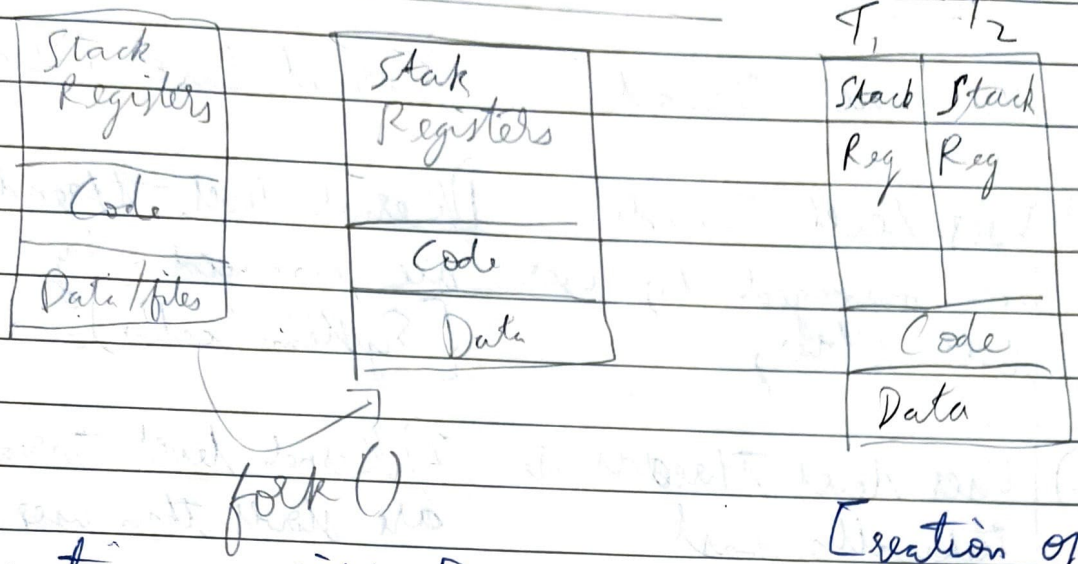


7 child ! parent

! No. of child processes created on writing fork() n times = $2^n - 1$

Total no. of processes created on writing fork() n times = 2^n

L-1.17 Process VS Threads in OS



Creation of child process

Creation of threads (Same code and data)

Process	Threads
1) System calls involved in Process	1) There is no system call involved
2) OS treats different process differently	2) All user level threads treated as single task for OS
3) Different process have different copies of data files, code	3) Threads share same copy of code and data
4) Content switching is slower	4) Content switching is faster
5) Blocking a process will not block another	5) Blocking a thread will block entire process
6) Independent	6) Interdependent

1/12

User Level Vs Kernel Level Thread

	User Level Thread	Kernel Level Thread
1)	User level threads are managed by user level library	1) Kernel level threads are managed by OS [System calls]
2)	User level threads are typically fast	2) Kernel level threads are slower than user level
3)	Context switching is faster	3) Context switching is slower
4)	If one user-level thread performs blocking operation, the entire process gets blocked	4) If one kernel level thread gets blocked, no effect on others

Similar to creating a process

Process > Kernel level Thread > User-level thread

Time taken