④ # System calls and its categorisation :-

⇒ System call is way to shift from user mode to kernel mode.

⇒ All os commands call this system call which done by kernel.

*** ⇒ Printf() is not a system call but is function: But printf internally makes system call to write() command on monitor. So this is the use of API to make system call. system call invokes kernel to perform this work.

⑤ System call (categorisation) :-

- → File related system call ⇒ open(); Read(), write(), Close(), Create file etc.
- → Device Related ⇒ Read, Write, Reposition, ioctl, fcntl.
- → Information ⇒ get Pid, attributes, get system time & date.
- → Process control ⇒ Execute, abort, *** fork, Wait, Signal, Allocate, Load etc.
- → communication ⇒ Create/delete connection., Pipe(), shmget()

⇒ Fork() → system calls create a child process of the process. And both parent and child Process keeps working simultaneously. so multiprocessing takes place here.

⇒ chmod → to change privileges to file.

⑤ # Fork() system call :- *(see it's video 6 for example question)

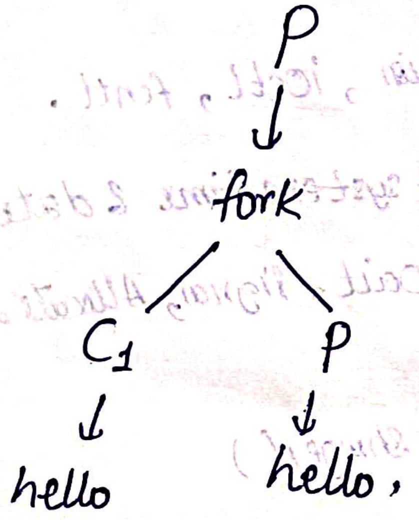
⇒ we use this to create child process.
or we could use thread to work in such thing.

⇒ we will see difference b/w thread & fork() system call.

⇒ Fork() → 0 → child (child process Id returns 0)
→ +1 → Parent(+ve) (Parents process Id returns +ve)
→ -1 → child x (not created). *(when child not created then returns -ve)
In worst case when Kernel is busy then my child is not created

⟹ e.g. code :-

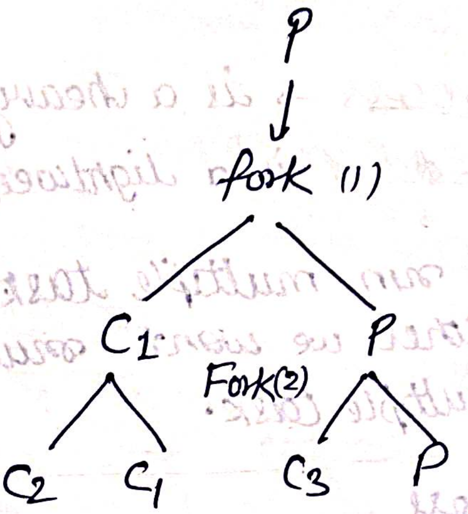
```
main() {
    fork();
    printf("hello");
}
```



So output ⟹ hello
hello

⇒ If we write fork two times then :-

```
main() {
|   fork(); (1)
|   fork(); (2)
|   print("hello");
}
```

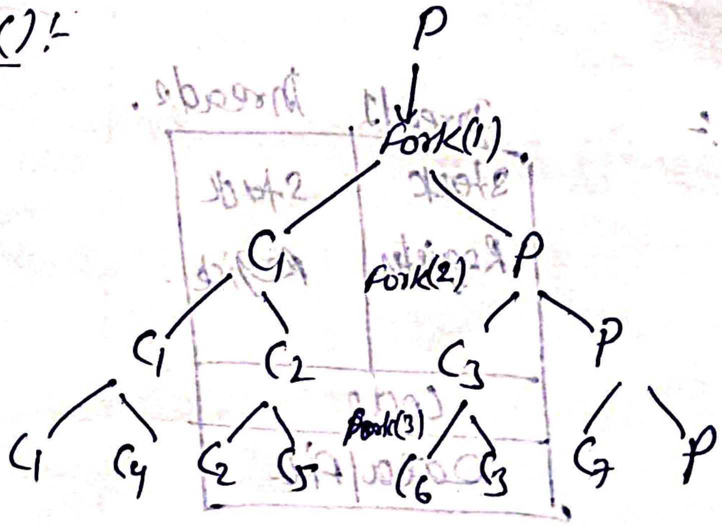


⇒ so here output ⇒

```
hello → C2
hello → C1
hello → C3
hello → P
```

⇒ so here we've 3 child process (C1, C2, C3) and 1 Parent Process.

⇒ For 3 times fork() :-



∴ Total 7 child process & 1 parent ∴

∴ 8 times hello will be printed.

$\boxed{\text{Hence total child process} \Rightarrow 2^n - 1}$ $\boxed{\text{total print} = 2^n}$

6/08/2

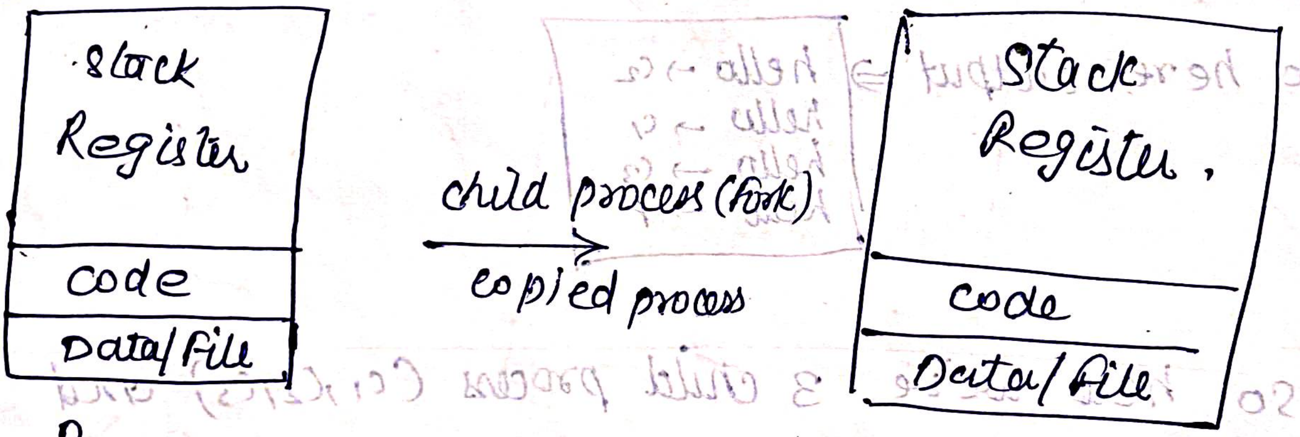
7 # Process Vs Threads :-

⇒ Here we are talking about multiprocess or multitask.

⇒ Process → is a heavyweight task
⇒ Thread → is a lightweight task.

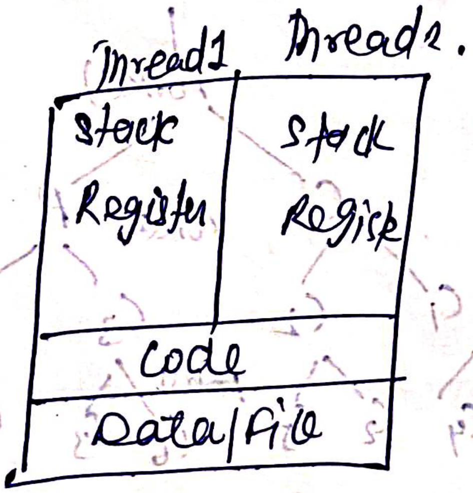
⇒ To run multiple task parallely in a single environment. either we want multiple CPUs or in single CPU perform multiple task.

Process

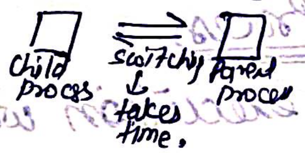


Process

8 Threads :-



(5) Difference :-

Process	Threads (user level)
(1) System calls involved in Process	(1) There is no system call involved
(2) Os treats different process differently. (Means child process (1 process) and Parent process is another process). It takes as two different process.	(2) All user level thread treated as single task for Os.
(3) Different process have different copies of Data, files, code. (child is copy of parent process)	(3) Threads share same copy of code and data.
* (4) context switching is slower  	(4) Content switching is faster. (Here shift or switch b/w registers, stack may take place).
* (5) Blocking a process will not block another process * (If parent blocks then it does not mean child will be also blocked) so they are independent.	(5) Blocking a thread will block entire process.
(6) Independent.	(6) Interdependent.

⑧ # User Level Thread	Kernel Level Thread
① user Level thread are managed by user level.	② kernel Level Threads are managed by OS (system call)
② user level threads are typically fast	② Kernel level threads are slower than user level.
③ Content switching is faster	③ Content switching is slower
④ If one user level through perform blocking operation then entire process get blocked.	④ If one kernel level thread blocked No affect on others.

Interview Question

⇒ Differen b/w Thread and Process

⇒ Thread is a path of execution within a process. ~~can~~

A process can contains multiple threads

⇒ The threads within the same process run in a shared memory space, while processes run in separate memory spaces.

Threads are dependent, while process are not.