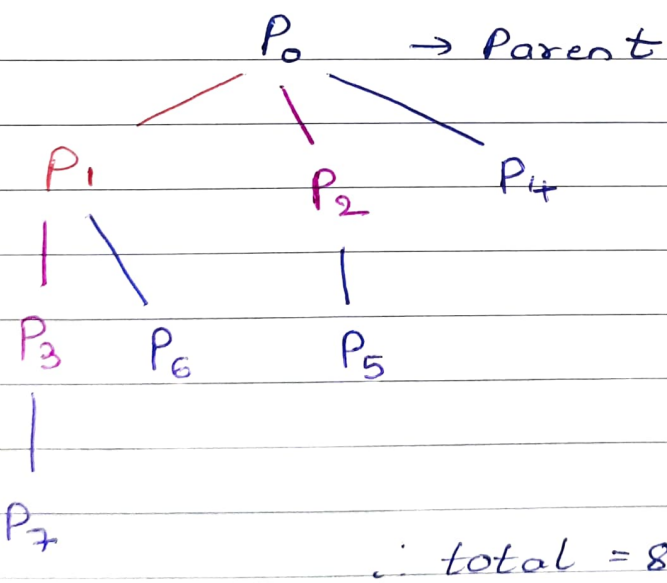
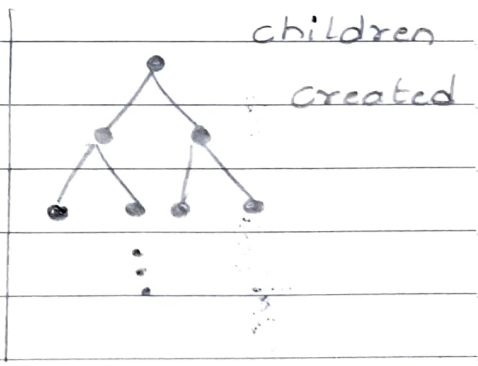
Count the number of times, "hello" is printed

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

fork(); fork(); fork();





∴ total = 8

Number of Processes = 2^n, where n is the number of fork() calls.

$n = 3$, $2^3 = 8$.

What will be the output of the following program:

```
#include <stdio.h>
#include <unistd.h>

void PC ()
{
        if (fork() ==0 )
            printf("child\n");
        else
            printf("Parent\n");
}

int main(){
   PC();
   return 0;
}
```

The output can be:

child
Paunt

OR

Parent
child

Both processes are running concurrently & we dont know whether OS will give control to parent process or child process.

prakash hegade

GATE Question:

A process executes the following code:

```
for(i=0; i<n; i++)
  fork();
```

The total number of child processes created is:

a) n
b) $2^n - 1$
c) 2^n
d) 2^{n-1}

Answer: b) $2^n - 1$

Write a program to achieve the following:
→ For a parent create 3 children & terminate as follows:
  → First, third child terminates
  → Second, Second child terminates
  → Third, First child terminates
  → Last, Parent terminates.

Parent & child process ids must be printed at every phase and in the end, print the parent id.

The termination sequence must be strictly as above.

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int pid1, pid2, pid3;

    pid1= fork();
    if (pid1 ==0)
    {
        sleep(3);
        printf(" child1 id = %d , Parent id = %d \n",
            getpid(), getppid());
    }
    else
    {
        pid2 = fork();
        if (pid2 ==0)
        {
            sleep(2);
            printf(" child2 id = %d , Parent id = %d \n",
                getpid(), getppid());
        }
        else
        {
            pid3 = fork();
            if (pid3 ==0)
            {
                printf(" child3 id = %d , Parent id = %d \n",
                    getpid(), getppid());
            }
```

```
    else
      {
        sleep(4);
        printf("Parent id = %d\n", getpid());
      }
    }
  }
  return 0;
}
```

Sample Output:

```
child3 id = 133 , Parent id = 130
child2 id = 132 , Parent id = 130
child1 id = 131 , Parent id = 130
Parent id = 130
```

Usage of exec and wait:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
                    → sys/wait.h

int main()
{
    pid_t pid;

    pid = fork();
    if(pid<0){
      fprintf(stderr, "Fork Failed");
      return 1;
    }
```

```
else if (pid == 0) {
    execlp ("/bin/ls", "ls", NULL);
}
```

↳ list terminated by NULL

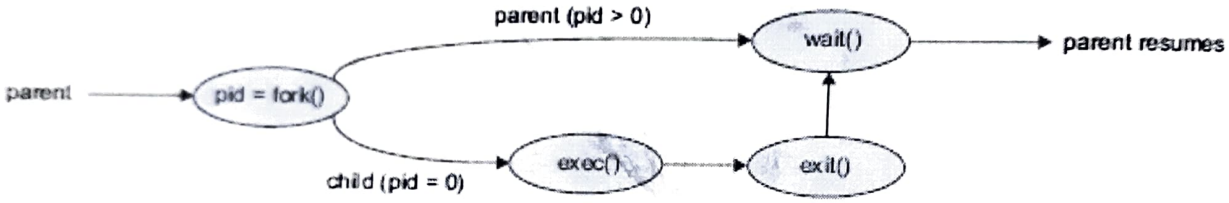
↳ command

↳ where to find command

```
else
{
    wait (NULL);
    printf("child Complete");
}
```

↳ parent will wait for the child to complete.

```
return 0;
}
```



Termination:

A parent may terminate the execution of one of its children for a variety of reasons such as,

- The child has exceeded its usage of some of the resources that it has been allocated
- The task assigned to child is no longer required
- The parent is exiting, and the OS does

not allow a child to continue if its parent terminates

Cascading termination:

- some systems do not allow a child to exist if its parent has terminated. In such systems if a process terminates, then all of its children must also terminate. This phenomenon is referred to as cascading termination.

exit()

↳ to terminate

Parent may terminate the execution of children processes using the -

abort() System call.

When the process terminates, normally or abnormally, the Kernel notifies the parent by sending the SIGCHLD signal to the parent.

Note: Ways to terminate:

1. Return from main 2. Calling exit 3. Calling _exit or _Exit 4. Return of the last thread from its start routine 5. Calling pthread_exit from the last thread 6. Calling abort 7. Receipt of a signal 8. Response of the last thread to a cancellation request.