Fork System Call

In multitasking OS, processes (which eventually run programs) might need a way to create new processes to run other programs & enhance the utilization of the system.

This is where Fork system call helps us.

It creates a copy of this original process & the copy is called a "child process".

1) Fork creates a separate physical address space for the child. The child has an exact copy of all memory segments of the parent.

∴ Both parent & child process has the same Program counter (PC) ∴ both point to the same next instruction

∴ We can say that both parent & child will begin their execution from the same next instruction following the fork.

In this example, at line 4) fork is invoked / called ∴ both parent & child can begin execution from line 5) i.e the next line / instruction after fork.

```
1) #include
2) int
3) .
   .
   .
4) fork()
->5)
```

2) File opened by the parent will be the same for child

3) Remember, the P_ID (unique ID) of child is obviously different from the parent P_ID

∴ henceforth remember, every process will have a P_ID (process ID) and PP_ID (parent process ID)

Now imagine a process having PID = 1214 calls fork & as a result a child is created. The PID of the child will be P_ID of parent + 1. ∴ P_ID of the child in this case will be 1215. ← Jmp

The PPID of the child = P_ID of its parent = 12 ↑ Jmp

(P)
PID = 1214
PP_ID = ?

(C)
PID = 1215
PP_ID = 1214

↳ we will see in the lab.

4) If 1 process invokes multiple fork, all P_ID of children will be unique (+1 rule applicable) but their PP_ID will be same.

∴

(P) PID = 1214
PPID = ?

(C1) PID = 1215, PPID = 1214

(C2) PID = 1216, PPID = 1214

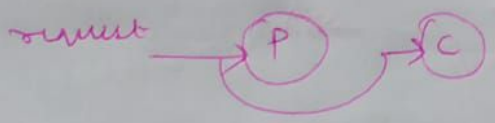
(C3) PID = 1217, PPID = 1214 . . . .

5) ∴ After a successful fork() call.

→ An integer value 0 is returned ⇒ successful child creation & value 0 is returned to the child

→ An integer value > 0 (nothing but the child P_ID) is returned to the parent.

→ An integer < 0 ⇒ fork was unsuccessful.



So a summary, Process P was running, he felt a need to create another process to delegate his work & thus he creates a child C using fork.

So if a request comes to P, he will delegate the request to C. Now C will do/serve this request & P will wait/do some other work.

∴ After fork, both C & P will work concurrently / simultaneously

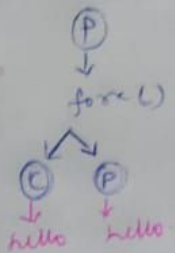
Q1)

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

→ at this stage a child will be created.
So we ultimately have both child & Parent.
We know that both will point to the next printf
∴ Both child & parent will print "Hello"
∴ output will be 2 Hello

Output: Hello
Hello //

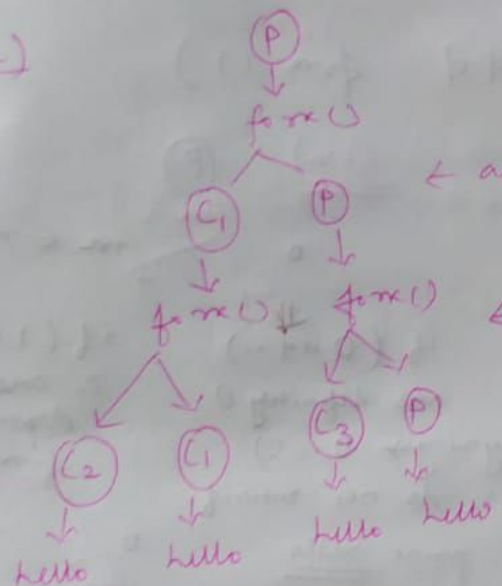
Visualization



Q2)

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

Soln)



← at this stage a child of the parent is created due to a) fork()

← Now both child & parent point to next instruction
fork() ∴ both will perform fork

↑ all will do next printf inst. "Hello"

Output: Hello
Hello
Hello
Hello //

∴ We can observe that for
1 Fork ⇒ 2^1 Hello.
2 Fork ⇒ 2^2 Hello etc.
∴ for n Fork, 2^n outputs
or 2^n processes created. Out of which $2^n - 1$ are child, The last one is the parent //

Q 3)

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

Solve

a) How many times is hello printed ?

b) Total no. of process ?

c) How many child process ?

d) How many parent process ?

Another imp. point we know till now is

fork() returns a value.

a) If that value = 0 ⇒ child

b) If that value is 1 or +ve basically > 0 ⇒ parent.

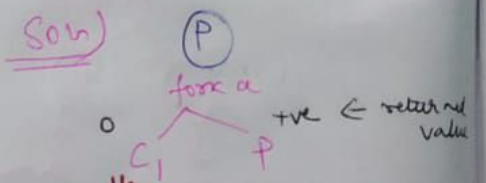
c) If that value < 0 or -ve ⇒ fork unsuccessful

(Remember, in future, by chance your child becomes a parent, he will have value +ve as now he is parent)

Q)

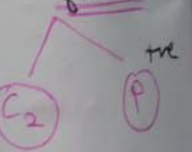
```
main()
{
if ( fork() (a) && fork() (b) )
{
fork(); (c)
}
printf("hello");
return 0;
}
```

Soln)



if (0 && ..)
ans definitely 0
(anything * 0 = 0)
∴ do not enter if
∴ print hello

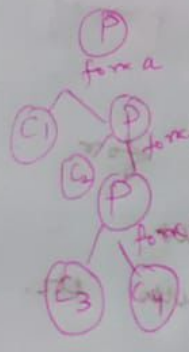
if (1 && ?)
∴ ans depends on fork b.



if (+ve && 0)
ans 0
∴ do not enter if
∴ print hello

Soln)

Output : hello
hello
hello
hello



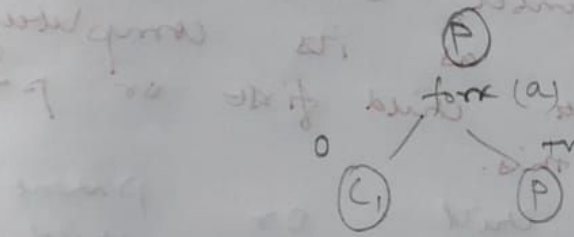
if (+ve && +ve)
∴ ans > 0
∴ enter if
& execute fork c

both C3 & C4 will execute hello

Total 4 hello printed

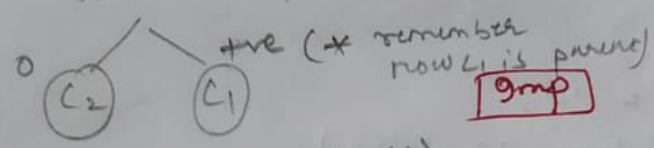
Try same a) with

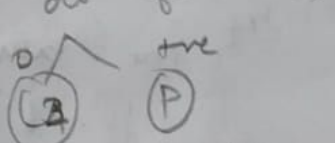
a)
```
main()
{
  if ( fork() (a)  ||  fork() (b) )
  {
    fork(); (c)
  }
  printf (" hello ");
  return 0;
}
```



if (0 || ?)
∴ execute fork (b)

if (1 || ?)
enter if definitely as one 1 for all case



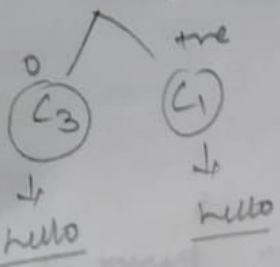


if (0||0)
do not enter if
print hello

if (0||1)
one +ve
∴ enter if
do fork(c)

print hello

print hello

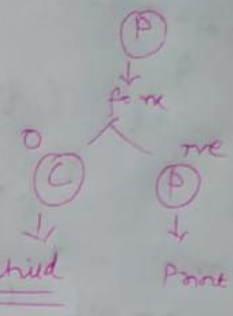


gmc

Output :
hello
hello
hello
hello
hello

Q)
```
int main()
{
  if (fork() == 0)
    {
      printf ("child");
    }
  else
    printf ("Parent");
}
```

Soln

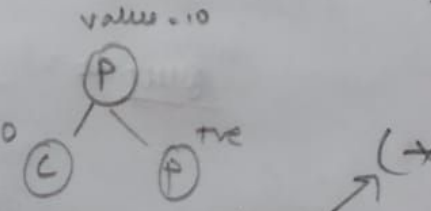


∴ Both words child & Parent printed depending on if or else part.

However remember, order of these 2 words depends on OS as its completely the choice of OS to execute child first or parent first user can't control this.

∴ Output) child parent Or parent child. //

Q)
```
int main()
{
  int value = 10;
  if (fork() == 0)
    {
      value += 5
      printf ("child = %d", value);
    }
  else
    {
      value -= 5
      printf ("Parent = %d", value);
    }
}
```



Imp
↓
remember child has copy of parent stuff ∴ they don't do not affect each other

value = 10
Enter if
value = 15
print child = 15

value = 10
Enter else
value = 5
print parent = 5

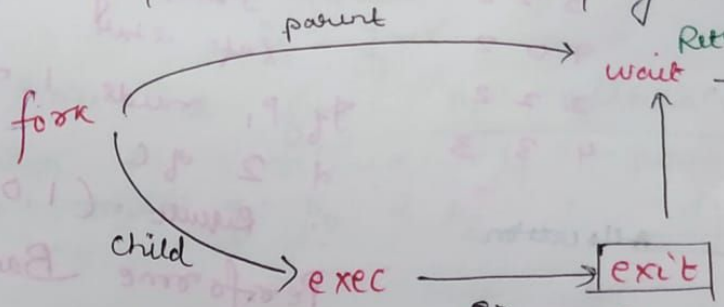
Output

Child = 15
Parent = 5 Or Parent = 5
Child = 15

Process Creation :-

fork :- System call to create new process

exec :- used after fork to replace the process memory space with a new program



- Process executes last statement and then asks OS to delete it using exit.
- returns status data from child to parent
- Process R are deallocated

abort :-

parent terminates execution of child

- child exceeds allocated R
- Task assigned to child is no longer required
- Parent exits OS.

If no parent waiting, did not wait invoked → Zombie (completed execution, still has entry in process table)

If parent terminated without invoking wait → Orphan
no parent still it continues
adopted by init process
↓
first process started during booting of computer system