Given a program on **fork()** system call.

3.9

```
#include <stdio.h>
#include <unistd.h>
int main()
{
   fork();
   fork() && fork() || fork();
   fork();

   printf("forked\n");
   return 0;
}
```

Run on IDE

How many processes will be spawned after executing the above program?

A *fork()* system call spawn processes as leaves of growing binary tree. If we call fork() twice, it will spawn 2^2 = 4 processes. All these 4 processes forms the leaf children of binary tree. In general if we are level **l**, and fork() called unconditionally, we will have $\mathbf{2^l}$ processes at level (**l+1**). It is equivalent to number of maximum child nodes in a binary tree at level (**l+1**).

As another example, assume that we have invoked fork() call 3 times unconditionally. We can represent the spawned process using a full binary tree with 3 levels. At level 3, we will have 2^3 = 8 child nodes, which corresponds to number of processes running.

A note on C/C++ logical operators:

The logical operator && has more precedence than ||, and have left to right associativity. After executing left operand, the final result will be estimated and execution of right operand depends on outcome of left operand as well as type of operation.

In case of AND (&&), after evaluation of left operand, right operand will be evaluated only if left operand evaluates to **non-zero**. In case of OR (||), after evaluation of left operand, right operand will be evaluated only if left operand evaluates to **zero**.

Return value of fork():

The man pages of fork() cites the following excerpt on return value,

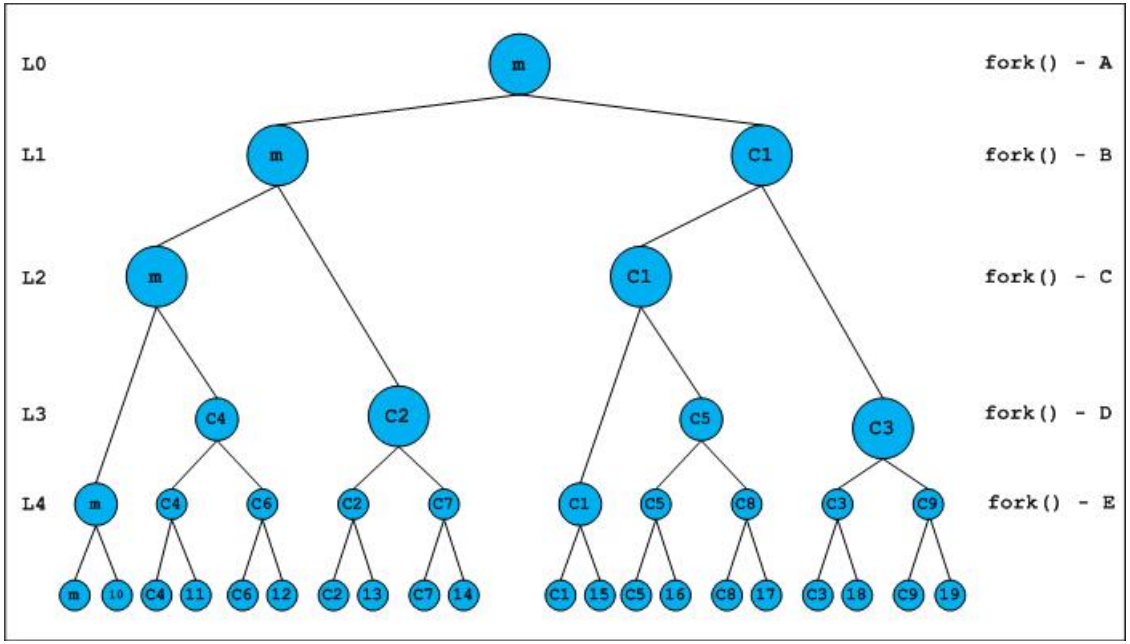
"*On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.*"

A PID is like handle of process and represented as *unsigned int*. We can conclude, the fork() will return a non-zero in parent and zero in child. Let us analyse the program. For easy notation, label each fork() as shown below,

```
#include <stdio.h>
int main()
{
   fork(); /* A */
   ( fork()  /* B */ &&
     fork()  /* C */ ) || /* B and C are grouped according to precedence */
   fork(); /* D */
   fork(); /* E */

   printf("forked\n");
   return 0;
}
```

The following diagram provides pictorial representation of fork-ing new processes. All newly created processes are propagated on right side of tree, and parents are propagated on left side of tree, in consecutive levels.



The first two fork() calls are called unconditionally.

At level 0, we have only main process. The main (m in diagram) will create child C1 and both will continue execution. The children are numbered in increasing order of their creation.

At level 1, we have m and C1 running, and ready to execute fork() – B. (Note that B, C and D named as operands of && and || operators). The initial expression B will be executed in every children and parent process running at this level.

At level 2, due to fork() – B executed by m and C1, we have m and C1 as parents and, C2 and C3 as children.

The return value of fork() – B is non-zero in parent, and zero in child. Since the first operator is &&, because of zero return value, the children C2 and C3 **will not** execute next expression (fork()- C). Parents processes m and C1 will continue with fork() – C. The children C2 and C3 will directly execute fork() – D, to evaluate value of logical OR operation.

At level 3, we have m, C1, C2, C3 as running processes and C4, C5 as children. The expression is now simplified to ((B && C) || D), and at this point the value of (B && C) is obvious. In parents it is non-zero and in children it is zero. Hence, the parents aware of outcome of overall B && C || D, will skip execution of fork() – D. Since, in the children (B && C) evaluated to zero, they will execute fork() – D. We should note that children C2 and C3 created at level 2, will also run fork() – D as mentioned above.

At level 4, we will have m, C1, C2, C3, C4, C5 as running processes and C6, C7, C8 and C9 as child processes. All these processes unconditionally execute fork() – E, and spawns one child.

At level 5, we will have 20 processes running. The program (on Ubuntu Maverick, GCC 4.4.5) printed “forked” 20 times. Once by root parent (main) and rest by children. Overall there will be 19 processes spawned.

A note on order of evaluation:

The evaluation order of expressions in binary operators is unspecified. For details read the post Evaluation order of operands. However, the logical operators are an exception. They are guaranteed to evaluate from left to right.

Contributed by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.