# The fork() System Call

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

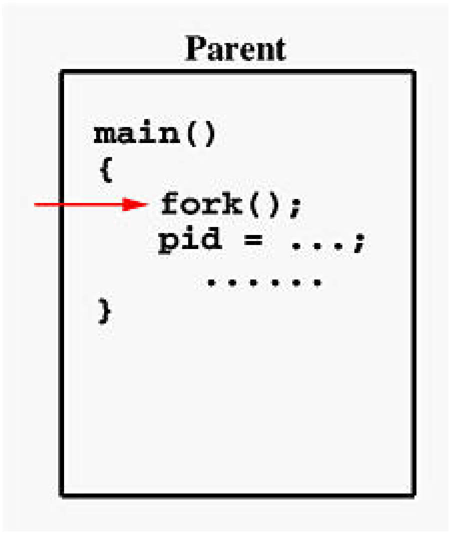
- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

Therefore, after the system call to **fork()**, a simple test can tell which process is the child. **Please note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces**.

Let us take an example to make the above points clear. This example does not distinguish parent and the child processes. Click **here** to download this file **fork-01.c**.

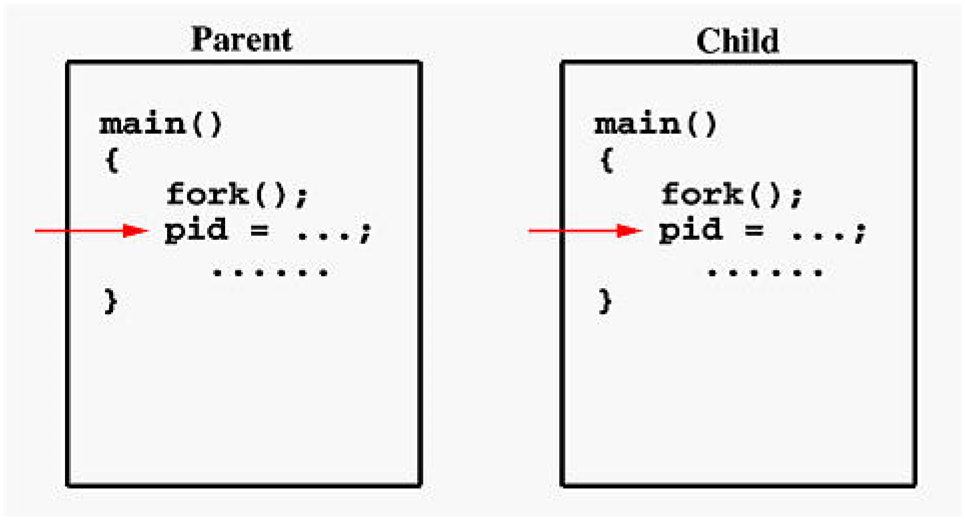
```
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  main(void)
{
     pid_t  pid;
     int    i;
     char   buf[BUF_SIZE];

     fork();
     pid = getpid();
     for (i = 1; i <= MAX_COUNT; i++) {
          sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
          write(1, buf, strlen(buf));
     }
}
```

Suppose the above program executes up to the point of the call to **fork()** (marked in red color):



If the call to **fork()** is executed successfully, Unix will

- make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the **fork()** call. In this case, both processes will start their execution at the assignment statement as shown below:



Both processes start their execution right after the system call **fork()**. Since both processes have identical but separate address spaces, those variables initialized **before** the **fork()** call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by **fork()** calls will not be affected even though they have identical variable names.

What is the reason of using **write** rather than **printf**? It is because **printf()** is "buffered," meaning **printf()** will group the output of a process together. While buffering the output for the parent process, the child may also use **printf** to print out some information, which will also be buffered. As a result, since the output will not be send to screen immediately, you may not get the right order of the expected result. Worse, the output from the two processes may be mixed in strange ways. To overcome this problem, you may consider to use the "unbuffered" **write**.

If you run this program, you might see the following on the screen:

```
     ................
This line is from pid 3456, value 13
This line is from pid 3456, value 14
     ................
This line is from pid 3456, value 20
This line is from pid 4617, value 100
This line is from pid 4617, value 101
     ................
This line is from pid 3456, value 21
This line is from pid 3456, value 22
     ................
```

Process ID 3456 may be the one assigned to the parent or the child. Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way. Moreover, the order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result.

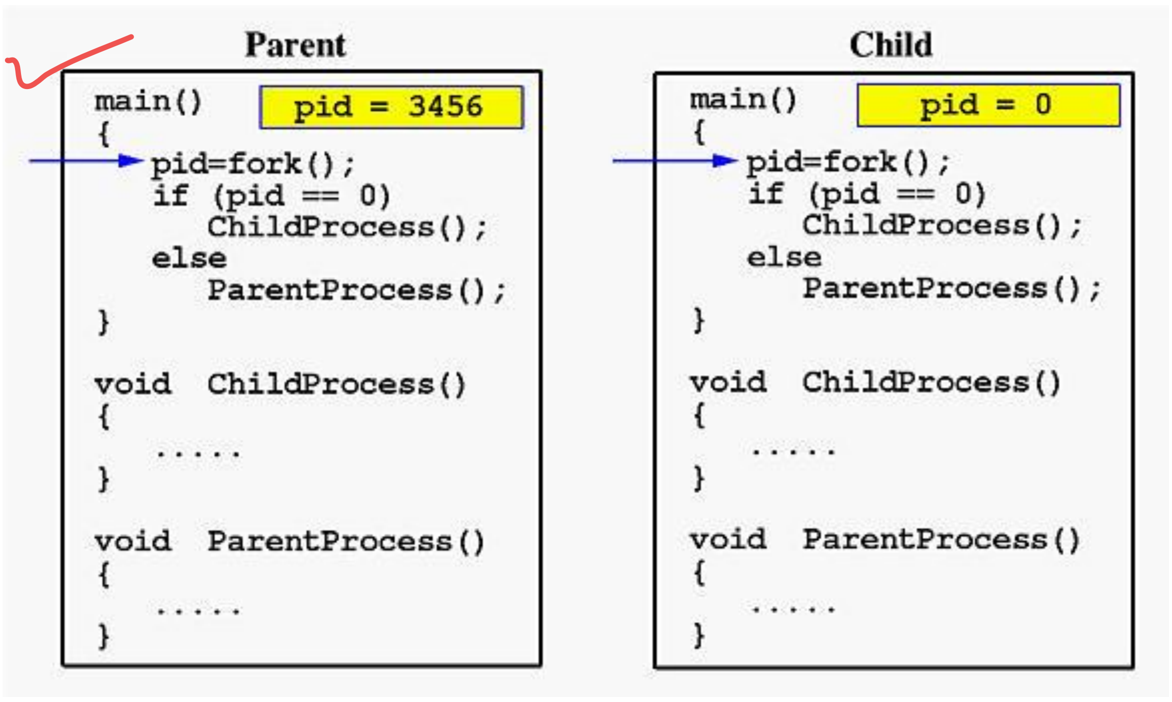
Consider one more simple example, which distinguishes the parent from the child. Click **here** to download this file **fork-02.c**.

```
#include  <stdio.h>
#include  <sys/types.h>

#define   MAX_COUNT  200

void  ChildProcess(void);                /* child process prototype  */
void  ParentProcess(void);               /* parent process prototype */

void  main(void)
{
     pid_t  pid;

     pid = fork();
     if (pid == 0)
          ChildProcess();
     else
          ParentProcess();
}

void  ChildProcess(void)
{
     int   i;

     for (i = 1; i <= MAX_COUNT; i++)
```

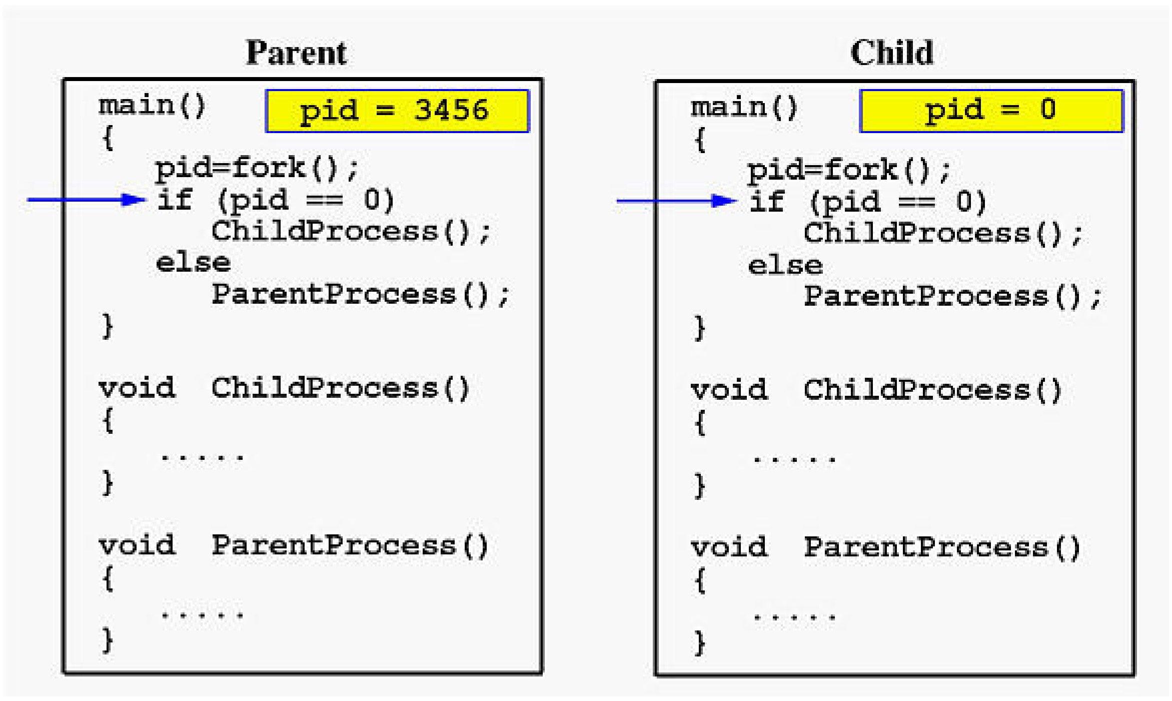
```
          printf("   This line is from child, value = %d\n", i);
     printf("   *** Child process is done ***\n");
}

void  ParentProcess(void)
{
     int   i;

     for (i = 1; i <= MAX_COUNT; i++)
          printf("This line is from parent, value = %d\n", i);
     printf("*** Parent is done ***\n");
}
```

In this program, both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and (2) the value of variable **i**. For simplicity, **printf()** is used.

When the main program executes **fork()**, an identical copy of its address space, including the program and all data, is created. System call **fork()** returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable **pid**. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.

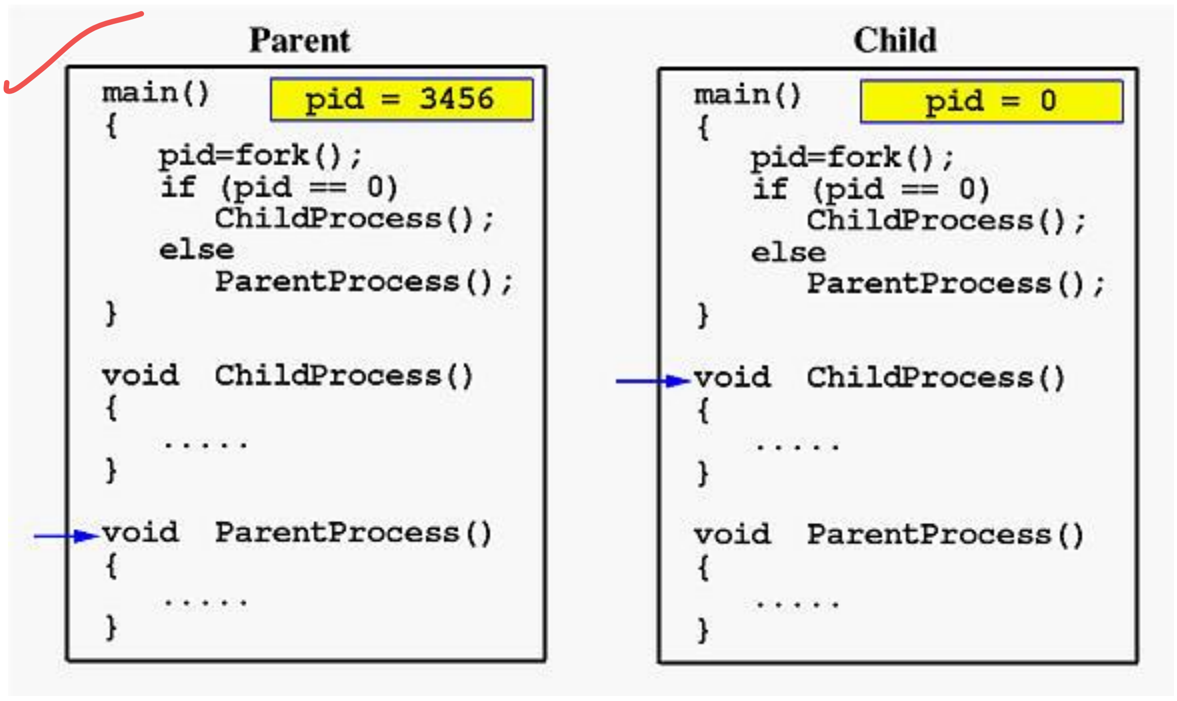


Now both programs (*i.e.*, the parent and child) will execute independent of each other starting at the next statement:



```
main()
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

void  ChildProcess()
{
   .....
}

void  ParentProcess()
{
   .....
}
```

In the parent, since **pid** is non-zero, it calls function **ParentProcess()**. On the other hand, the child has a zero **pid** and calls **ChildProcess()** as shown below:



```
main()
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

void  ChildProcess()
{
   .....
}

void  ParentProcess()
{
   .....
}
```

Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process. Therefore, the value of **MAX_COUNT** should be large enough so that both processes will run for at least two or more time quanta. If the value of **MAX_COUNT** is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process.

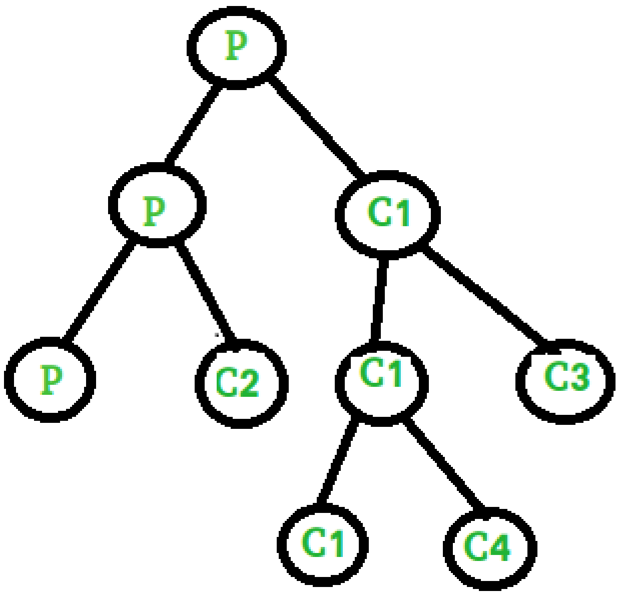
## Greekforgreeks:

Prerequisite: basics of fork, fork and binary tree, Example1: What is the output of the following code?

```
#include <stdio.h>

#include <unistd.h>

int main()

{

    if (fork() || fork())

        fork();

    printf("1 ");

    return 0;

}
```

Output:

```
1 1 1 1 1
```

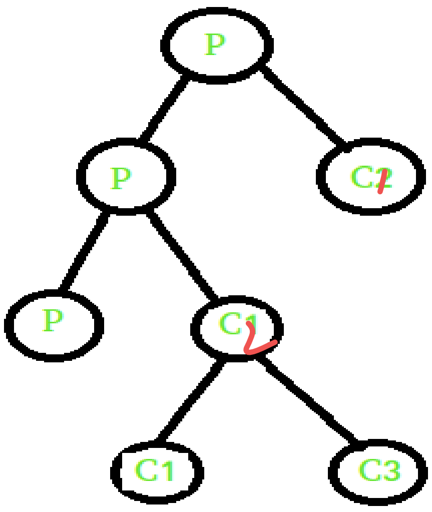


Explanation: 1. It will create two process one parent P (has process ID of child process)and other is child C1 (process ID = 0). 2. In if statement we used OR operator( || ) and in this case second condition is evaluated when first condition is false. 3. Parent process P will return positive integer so it directly execute statement and create two more processes (one parent P and other is child C2). Child process C1 will return 0 so it checks for second condition and second condition again create two more processes(one parent C1 and other is child C3). 4. C1 return positive integer so it will further create two more processes (one parent C1 and other is child C4). Child C3 return 0 so it will directly print 1. Example 2: What is the output of following code?

```
#include <stdio.h>

#include <unistd.h>

int main()

{
     1
  if (fork()) {
           0
     if (!fork()) {
            fork();

            printf("1 ");

        }

        else {

            printf("2 ");

        }

    }

    else {

        printf("3 ");

    }

    printf("4 ");

    return 0;

}
```

Output:

```
2 4 1 4 1 4 3 4
```

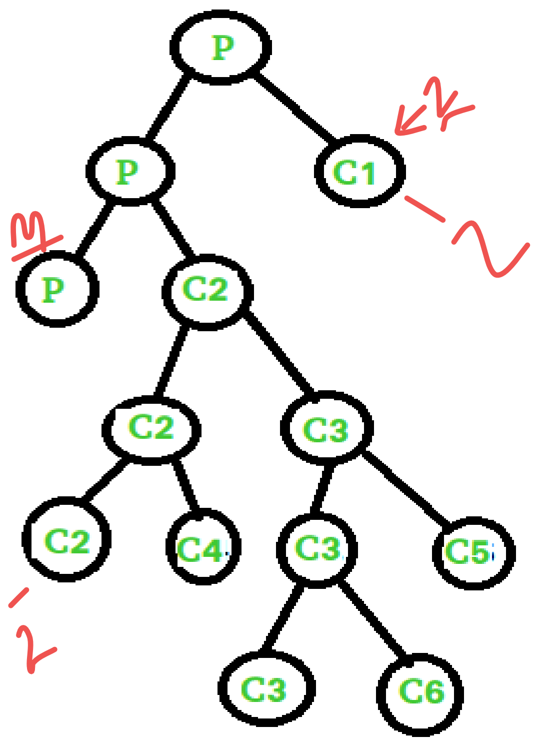


Explanation: 1. It will create two process one parent P (has process ID of child process) and other is child C1 (process ID = 0). 2. When condition is true parent P executes if statement and child C1 executes else statement and print 3. Parent P checks next if statement and create two process (one parent P and child C2). In if statement we are using not operator (i.e, !), it executes for child process C2 and parent P executes else part and print value 2. Child C2 further creates two new processes (one parent C2 and other is child C3). Example 3: What is the output of following code?

```
#include <stdio.h>

#include <unistd.h>



int main()

{

   if (fork() && (!fork())) {

      if (fork() || fork()) {

         fork();

      }

   }

   printf("2 ");

   return 0;

}
```

Output:

```
 2 2 2 2 2 2 2
```



Explanation: 1. Fork will create two process one parent P (has process id of new child) and other one is child C1 (process id=0). 2. In if statement we are using AND operator (i.e, &&) and in this case if first condition is false then it will not evaluate second condition and print 2. Parent process P check for second condition and create two new processes (one parent P and other is child C2). In second condition we are using NOT operator which return true for child process C2 and it executes inner if statement. 3. Child C2 again create two new processes (one parent C2 and child C3) and we are using OR operator (i.e, ||) which evaluate second condition when first condition is false. Parent C2 execute if part and create two new processes (one parent C2 and child C4) whereas child C3 check for second condition and create two new processes (one parent C3 and child C5). 4. Parent C3 enters in if part and further create two new processes (one parent C3 and child C6).