Dextutor


```
pid_t p;
printf("before fork\n");
p=fork();
if(p==0)
{
printf("I am child having id %d\n",getpid());
```

Program for fork() system call

2 Comments / Programs, System calls / By Baljit Singh Saini

fork() is a system call used to create a **new** process. The new process is called a child process and the original process is called the parent process. The child process by default is a duplicate of the parent process. By duplicate we mean that the child process has the same code as the parent process but the memory space of both the processes is separate. The syntax of using fork() is :

Syntax of fork() system call

```
#include<unistd.h>

pid_t fork(void);
```

fork() returns -1 on failure; On success it returns '0' in the child process and process-id of the child in the parent process.

What is the need to duplicate a process?

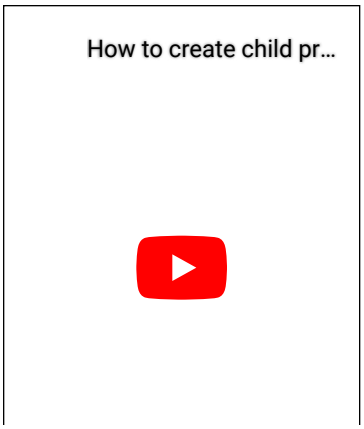
It may happen that a process is required to do two tasks that are independent. Since the tasks are to be done by the same process, they can be executed one after the other only. Now to do the same task in roughly half the time the process (parent) creates a new process (child). One of the task is performed by the parent and the other by the child. Consider it as a situation where you are supposed to do to two tasks and to complete them in quick time you take help of your friend which does one of the task for you.

We know that a process is program in execution. For the parent process the program is written by the programmer but from where the child gets its program? The child runs the same code as its parent, hence, it is called a duplicate process.

If the same code is run both by the parent and the child then isn't the task done twice? It's the job of the programmer to write the code in such a manner that only one of the task is done when the parent process runs the code and the other task is done when the child runs the code.

Program for fork() system call

```
{
pid_t p;
printf("before fork\n");
p=fork();
if(p==0)
{
printf("I am child having id %d\n",getpid());
printf("My parent's id is %d\n",getppid());
}
else{
printf("My child's id is %d\n",p);
printf("I am parent having id %d\n",getpid());
}
printf("Common\n");
}
```



Output:

```
baljit@baljit:~/Programs$ gcc fork.c
baljit@baljit:~/Programs$ ./a.out
before fork
My child's id is 28
I am child having id 28
I am parent having id 27
My parent's id is 27
common
common
```

Working of fork() system call:

After compiling the program with gcc it creates an output file "a.out". The moment you run a.out using the command, ./a.out, a new process is created (parent). This process gets the process id (PID) 27. The PID will differ from system to system and each time you run the program. The process starts to run and it prints **before fork**. Next it executes the fork() system call. If it gets executed a child process is created having a different PID. Now there are two process in the system both having the same code to run. But since the code has been run till this line the execution will continue from the next line in both the process. fork() on success returns either 0 or a non-zero value. Since, the same code is both the processes the variable 'p' will have some value in both the process. In the parent process it gets a non-zero positive value (which actually is the PID of the child). In the child process 'p' gets the value '0'.

I am parent having id 27

Are printed by the parent process.

When the if-else case runs from within the child the if condition becomes true and hence the lines

I am child having id 28

My parent's id is 27

Are printed by the child process.

Since the printf("common\n") line was out of the if-else it has been printed twice; once by the parent process and once by the child process.

By analysing the PID printed by each process the parent-child relationship can also be verified between them. The process with PID 27 says its child has a PID 28. Similarly, the process with PID 28 says its parent has a PID 27.

Note: The function getpid() is used to print the PID of a process while the function getppid() is used to print the PID of the parent process.

Point to Remember:

Whatever task you want the parent to perform should be written in the *else* part and the task for the child process should be written in *p==0* part. Anything outside the if-else will be performed by both parent and child.

Why the output is not in an order?

If the code is run multiple times, the order of the output lines may differ. Since there are two processes, the processor can be assigned to any of them. One of them can pre-empt the other and hence the output may overlap. The order of the output will differ. If you want the parent to execute first or the child to execute first then either wait() or sleep() functions can be used depending on the requirement.

Practice Program on fork() system call

Q1. Write a program using fork() system call to create two child of the same process i.e., Parent P having child process P1 and P2.
Q2. Write a program using fork() system call to create a hierarchy of 3 process such that P2 is the child of P1 and P1 is the child of P.

Viva questions on fork() system call

Q1. What does the fork() system call return on success?
Q2. What is the PID of the child process?
Q3. Which function is used to get the PID of a process?
Q4. How many total process are created with the below code

```
int main()
{
  fork();
  fork();
}
```

Relevant Programs

- Program for open() system call
- Program for creating threads
- Program for IPC using mkfifo()

2 thoughts on “Program for fork() system call”

RAMESH
MAY 13, 2020 AT 2:47 PM

Is the answer to Q4 “2”

Log in to Reply

BALJIT SINGH SAINI
MAY 13, 2020 AT 9:58 PM

NO..Total number of processes in the system will be 4 and the child process will be 3

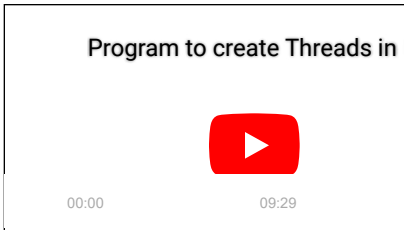
Log in to Reply

Leave a Comment

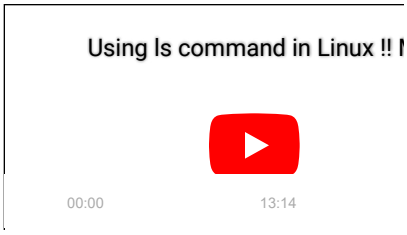
You must be logged in to post a comment.

Search ...

OS Lab



Linux Essentials Series



OS Theory

Categories

- Database
  - Entity-Relationship(ER)
  - Introduction to Database
- Linux
  - commands
  - Shell Scripting
- Operating System
  - CPU Scheduling
  - Deadlock
  - Disk Management
  - File Management
  - Introduction
  - IPC
  - Memory Management
  - Practice Problems
  - Process
  - Process Synchronization
  - System calls
  - Thread
- Programs
- Question Hub