National University of Computer and Emerging Sciences

Operating System Lab - 04

Lab Manual

Objective

The objective of this lab is to learn about Linux process management, commands to see running processes, system call to create processes, get their IDs, and wait and exec system call and to learn about zombie and orphan processes.

Process Management

In general, a process is an instance of a program written and compiled. There can be multiple instances of a same program. In other words, a program that is loaded into computer's memory and is in a state of execution is called a process. Processes are dynamic entity. Process essentially requires CPU and RAM resources but may also require I/O, Network or Printer depending on the program written.

Linux Processes

A Linux process or task is known as the instance of a program running under Linux environment. This means that if 10 users from servers are running gedit editor, then there are 10 gedit processes running on the server. Although they are sharing same executable code. The processes running in a Linux system can be view using 'ps' command which we covered in *lab manual 02.*

Process ID

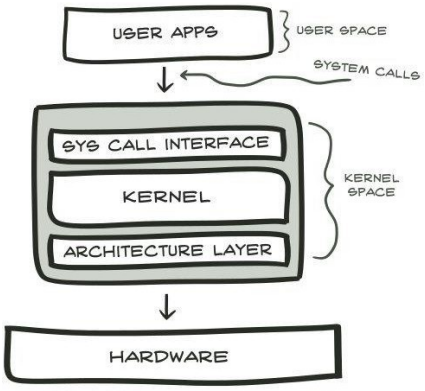
In Linux system, each process running has been assigned a unique ID which is known as PID (Process Identification Number). For example, Firefox is a running process if you are browsing the internet. Each time you start a new Firefox browser the system will automatically assign a PID to the new process of Firefox. A PID is automatically assigned when a new process is created on the system. If you wish to find out the PID of the running process like Firefox you may use the command 'pidof'.

```
$ pidof firefox

$ pidof bash

$ pidof bioset
```

To view the running process in a form of a tree we can use 'pstree' command. Which shows the running process in a form of a tree.

System Call Related to Process

There are a lot many Linux system calls available inbuilt for the users. A system call is as good as a function in C Programming Language. It can be compared with 'printf' function in C. The reason that it is often called system call rather than functions is that functions are limited to programming while system calls are specific for operating systems.



Fork System Call

In Linux, process is created by duplicating parent process. This is called forking. You invoke the fork system call with fork() function.

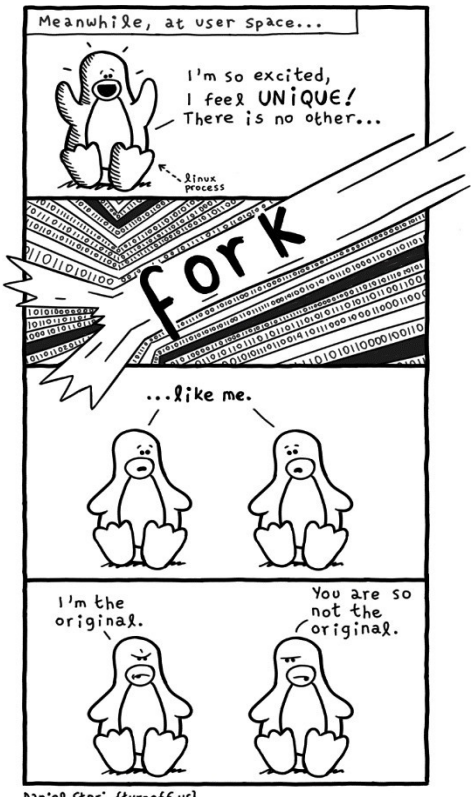
Definition

It is a system call that creates a new process under Linux operating system. It takes no argument. The purpose of fork() is to create a new process which becomes the child process to the caller. After the new child process is created, both processes will execute next instruction following the fork system call, therefore we have to distinguish the parent process from the child which can be done by evaluating the returned value of fork() function.

Returned Value from Fork

By examining the returned value of fork function, we can determine the current process is parent or child process. The following are the returned value and their meaning.

- If the returned value is negative, it means that the child process creation was unsuccessful
- If the returned value is zero, the child process is created with pid = 0
- If the returned value is positive, the child process is created with the process with a process ID to the parent process (The returned process ID is of type pid_t defined in sys.type.h). Normally this process ID is an integer.



After the system call to fork() is issued, a simple test can tell which process is the child. Note that Linux will make an exact copy of the parent address space and give it to the child. Therefore, the parent and child processes have separate address space.

Example

Consider the following C language program:

```c
#include<stdio.h>
#include <sys/types.h>
#include<unistd.h>
int main() {
printf("before forking \n");
printf("creating child process\n");
int i= fork(); // child process will be created
// and it will execute the program after fork.
if (i==0)
{
printf("I am child process\n");
}
else
{
printf("I am parent process\n");
}

printf("after forking \n");
return 0;
}
```

Implement the above code and check the output.

Compilation of program:

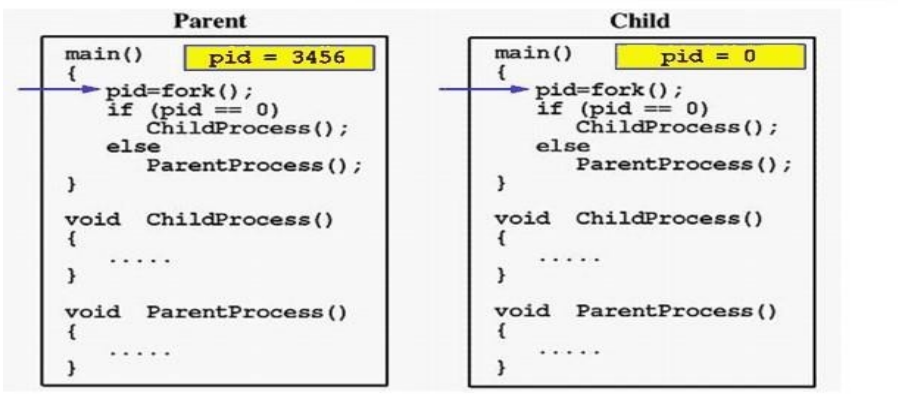
gcc -o obj filename.c

./obj

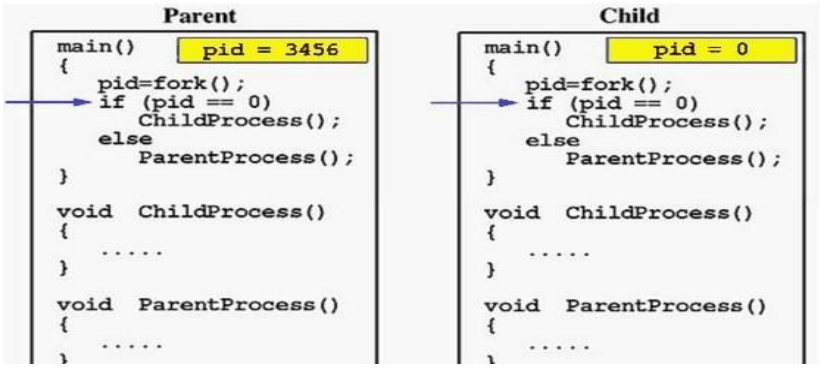
Example for 'fork()' Distinguishing Parent and Child

I 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.

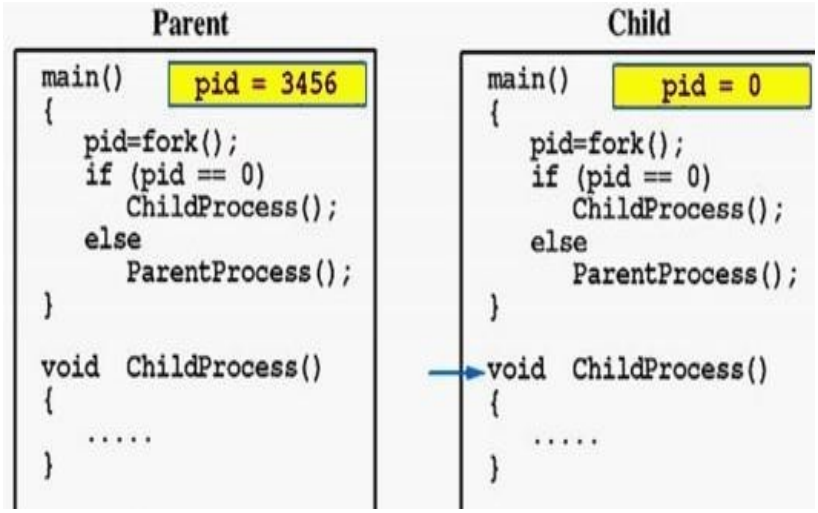
The images below will diagrammatically show the above code execution of forking:



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



In the above image, the fork function is called, which creates a new process and assign value of pid in parent process and assign 0 in child process. During if condition, it checked if the pid is zero or not to distinguish between parent and child process and invoke the functions of each process respectively.



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. The following codes distinguishes parent and child process working on variable in shared and separate address spaces:Even though the variable name remain same in child process yet due to separate address of process the variable is considered a new variable in process.

```c
#include<stdio.h>
#include <sys/types.h>
#include<unistd.h>
void parent_process(int cvar);
void child_process(int pvar);

int y=10;
int main() {
int x=0;
printf("before forking \n");
printf("creating child process\n");
int i= fork();
if (i==0)
{
child_process(x);
}
else
{
parent_process(x);
}
printf("after forking \n");
return 0;
}
void child_process(int a){
y+=2;
a=3;
printf("the value of child process variable= %d\n",a);
printf("In child process: y=%d\n",y);
}
void parent_process(int b){
b=2;
y+=5;
printf("the value of parent process variable= %d\n",b);
printf("In Parent process: y=%d\n",y); }
```

Implement the above program and check the output:

System Call to Get Process IDs

There are two system calls (functions) which can get the Process ID one is to get the process ID of the current process and another one to get the ID of its parent process. These are:

- 'getpid()' returns the PID of current process
- 'getppid()' returns the PID of parent's process

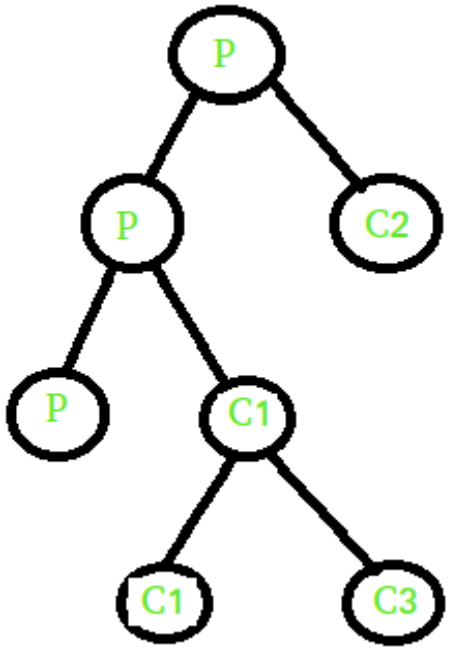
Task: *In the above example print the process ID of parent and child process and their parent processes plus child process should also print his her grand parent process id.*

Multiple child process:

In the following example multiple child are created . The execution sequence may be different in your system as compare to the commented sequence of print.

Execute the program and verify the creation of multiple child. Understand the tree.

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

int main()
{
    if (fork()) {
        if (!fork()) {
            fork();
            printf("1 ");//5,7
        }
        else {
            printf("2 ");//1
        }
    }
    else {
        printf("3 "); //3
    }
    printf("4 ");//2 ,4,6,8
    return 0;
}
```



[Different Kinds of Child Process]

[Zombie Process]

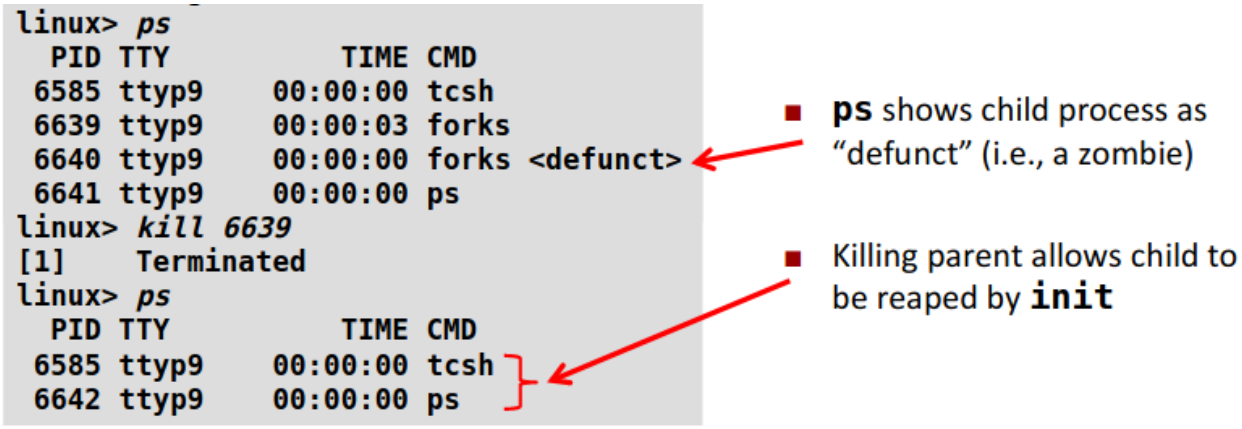
Reaping Child Processes

- **Idea**
  - When process terminates, it still consumes system resources
    - Examples: Exit status, various OS tables
  - Called a "zombie"
    - Living corpse, half alive and half dead
- **Reaping**
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process
- **What if parent doesn't reap?**
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by **`init`** process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

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

void fork7() {
if (fork() == 0) {
/* Child */
printf("Terminating Child, PID = %d\n", getpid());
exit(0);
}
else {
printf(" Running Parent, PID = %d\n", getpid());
while (1);
/* Infinite loop */
} }
int main()
{
printf("hello from main\n");
fork7();
return 0;
}
```

Implement the above program and check the output by opening two terminals. In one terminal, you need to execute your zom.c program and in second terminal execute ***ps -al*** to check the status of child process. As given below.

Then find the parent process id and kill the child process.



Solution to Avoid Creation of Zombie Processes

Using wait() system call we can avoid the creation of zombie process. The below changes to the above code will force the parent to wait:

Parent without WAIT();

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


int main() {
int pid = fork();
if(pid > 0) {
sleep(10);
printf("\n Parent");
printf("\n PID is %d",getpid());
}
if(pid == 0)
{
printf("\n Child");
printf("\n PID is %d", getpid());
printf("\n Parent PID is %d", getppid());
}
return 0;
}
```

Parent with WAIT():

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


int main() {
int pid = fork();
if(pid > 0) {
wait(NULL);
sleep(10);
printf("\n Parent");
printf("\n PID is %d",getpid());
}
if(pid == 0)
{
printf("\n Child");
printf("\n PID is %d", getpid());
printf("\n Parent PID is %d", getppid());
}
return 0;
}
```

What happens when the main make-zombie program ends when the parent process exits, without ever calling wait? Does the zombie process stay around? No—try running ps again, and note that both of the make-zombie processes are gone. When a program exits, its children are inherited by a special process, the init program, which always runs with process ID of 1 (it's the

first process started when Linux boots).The init process automatically cleans up any zombie child processes that it inherits. Implement the above program and check the output.

Orphan Process:

In general English terms, orphan is someone who lost parents. Same is the story here. If the parent process has finished execution and exited, but at the same time if the child process remains UN-executed, the child is then termed to be an orphan. This is done by making the child process sleep for sometimes. By that time of child's sleep, parent process will complete its execution and will exit. Since, parent process is no more there, child is referred to be an orphan now.

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

int main() {
int pid = fork();
if (pid > 0) {
//parent process
sleep(1);
printf("\n Parent ");
printf("\n PID is %d", getpid());
}
if(pid == 0) {
sleep(5);
printf("\n Child ");
printf("\n PID is %d",getpid());
printf("\n Parent PID is %d",getppid());
}
wait(NULL);
return 0;
}
```

Implement the given below program and check the child process with ps -al command.

Task: *As you do not see child process therefore, us wait system call to the program and again check the successful execution and return of child to parent.*

Execute System Call 'exec'

Fork allow you to create a process by duplicating the process of the current program, that is from whom it is called. However, limitation arises when you wish to execute a different program and this is where execute system call come in handy!

The **exec()** family of functions creates a new process image from a regular, executable file. This file is either an executable object file, or an interpreter script. There is no return from a successful call to an exec() function, because the calling process is functionally replaced by the new process.

The arguments specified by a program with an exec() function are passed on to the new process image in the corresponding main() arguments.

int execl(const char *_path_, const char *_arg0_, .., const char *_argn_, (char *)0);

path: Specifies the path name of the new process image file.

arg0, ..., argn: Point to null-terminated character strings. These strings constitute the argument list for the new process image. The list is terminated by a NULL pointer. The argument *arg0* should point to a file name that is associated with the process being started by the exec() function.

Last argument: is passed null as the function passes string.

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

int main() {
execl("/bin/ls", "ls", (char*)0);
/* We can only reach this code
if execl returned with
* an error
*/ perror("execl");
 return 0;
}
```

Execute the above code , you should get output of current directory files.

To avoid the replacement of current process, we can use fork and create child process which will be executing the other execution units.

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int PID;
  char cmd[256];
        Printf("Press e if you want to terminate\n");
  while (1) {
    printf("cmd: ");
    scanf("%s",cmd);
    if ( strcmp(cmd,"e")==0) /* loop terminates if type 'e'*/
      exit(0);
    /* creates a new process. Parent gets the process ID. Child gets 0 */
    if ((PID=fork()) > 0)
      wait(NULL);
    else if (PID == 0) /* child process */
      {
        execlp (cmd,cmd,NULL);
        /* exec cannot return. If so do the following */
        fprintf (stderr, "Cannot execute %s\n", cmd);
        exit(1); /* exec failed */
      }
    else if ( PID == -1)
      {
        fprintf (stderr, "Cannot create a new process\n");
        exit (2);
      }
  }

}
```

Execute the above example and find the output, understand how it works.