OPERATING SYSTEMS LAB

LAB MANUAL # 06

Instructor: Muhammad Abdullah Orakzai

Semester Fall 2021

Course Code: CL2006

Fast National University of Computer and Emerging Sciences, Peshawar

Department of Computer Science

Contents

Operating Systems Processes

OS Process Basic Terminologies

Process: Program in running state.

For example, as we know that MS word is a program when you run MS word in your computer.

This state of program is called running state. Now this running state of program is called process.

PID: Process ID assigned to process by default.

PPID: Parent Process ID.

INIT: PID→1 →started by kernel→No parent process. INIT is a process started by kernel. Init is the parent of all the tasks/processes. This Init process is started by OS.

DAEMON: Process start up at booting and keep running forever in background. Some processes are running continuously in our computer till your computer is on and stops on shutdown your computer. Some programs are running continuously (in background) in computer when your computer is switched on and stops on computer is switched off. A daemon is a long-running background process that answers requests for services.

Daemon has no user interface. (In window and in android it is called services).

Some examples include inetd, httpd, nfsd, sshd, named, and lpd

Kill: You want a process to die.

Zombie: Killed process still showing on system. After killing process, it will not be running but will be showed (its name will be visible) then will become zombie. You cannot kill zombie again because it is already killed once.

Processes

System Call

A system call is an interface between your program and the kernel. The Linux kernel's job is to provide a variety of services to application programs this is done using the provision of system calls.

We communicate with OS with the help of system calls.

Understanding Processes

The fundamental building block of each program is the process. A process is the name given to a program when it is loaded “for” the purpose of execution on the operating system. For a full description of what is comprised “in-side” a process, please refer to your class-notes or slides.

To view a list of current processes in the system for a user, you may use the ***ps* command**. **ps** shows tasks or processes running on current terminal. Here bash is running which is your shell and ps was itself running that told that what process are running.

ps x

ps x | less

ps aux | less

ps aux | more

ps aux: Shows running processes of the whole system. In Linux the command: **ps** -**aux**. Means show all processes for all users. You might be wondering what the **x** means? The **x** is a specifier that means '**any of the users'**.

ps fax | less

pgrep netns (netns is a process name and this command will show process id)

more about ps command

man ps

top command: is used to see programs that are running. Write simply top on terminal i.e.

top

press q to exit

Task and **process** are interchangeably used.

To view the complete list, you can adapt it to:

ps au

```
muhammad@muhammad-VirtualBox:~$ ps au
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
muhammad    1408  0.0  0.1 172988  6600 tty2     Ssl+ 21:14   0:00 /usr/lib/gdm
muhammad    1411  1.0  1.7 835988 70404 tty2     Sl+  21:14   0:25 /usr/lib/xor
muhammad    1463  0.0  0.3 197388 14040 tty2     Sl+  21:14   0:00 /usr/libexec
muhammad    3221  0.0  0.1  19720  5064 pts/0    Ss   21:51   0:00 bash
muhammad    3241  0.0  0.0  20468  3520 pts/0    R+   21:56   0:00 ps au
```

You will see plenty of columns as output. The columns you should be familiar with at this moment are underlined below:

- **User:** The owner of that process.
- **PID:** The integer identifier.
- **CPU:** Percent utilization of CPU.
- **MEM**: Percent utilization of Memory space.
- **VSZ:** Virtual Memory Size.
- **RSS:** Non-swapped physical memory size.
- **TTY:** Controlling Terminal.
- **STAT:** The process states (D) Uninterruptible sleep (R) Running or ready (S) Interruptible sleep (T) Stopped (Z) Zombie (<) High Priority (N) Low Priority (s) Session leader, i.e., has child processes (l) multi-threaded (+) foreground.
- **START:** When process has started.
- **TIME:** Time running since.
- **COMMAND:** The program/command used in this process.

The basic attribute of a process is it's ID (PID), and its Parent ID (PPID). The system calls that can find the process ID, and the Parent Process ID are the getpid() and getppid() calls respectively. To use both of them, you will be required to include the ***sys/types.h*** and ***unistd.h*** C libraries.

The mere presence of the PPID means that there is a hierarchy of processes inside our operating system. Each process has a track of who are its children, and each child process has a record of who is its parent. The original process that is the grand-father of all processes is the **Init** Process. You can view this hierarchy using the **pstree** command.

pstree Command

This command will show a list of all processes currently in the system in the form of a tree.

systemd is a suite of basic building blocks for a Linux system. It provides a system and service manager that runs as PID 1 and starts the rest of the system.

pgrep systemd Will find PID of this process.

Exercise 1

With respect to the myfirst.c file that you created in last labs, write a code that is able to find out the following:

- The PID value for myfirst.c.
- The PPID value for myfirst.c.
- **The process name from the PPID value**. (Note: You can use the system() call for this purpose. To find a process name from PPID, you would normally enter the following command on the shell, where 2419 is the PPID of any process).

pstree -p | grep 2419

C program to get Process Id and Parent Process Id in Linux

This program will ***get the Process Id and Parent Process Id of the current Process*** in C programming Linux.

Here we are using two functions **getpid()** to get Process Id and **getppid()** to get Parent Process Id of the **current process**. These functions are declared in <unistd.h> header file.

processId.c → (Program name**)**

```
/*C program to get Process Id and Parent Process Id in Linux. */

#include <stdio.h>

#include <unistd.h>

#include<stdlib.h>

int main()

{

   int p_id,p_pid;

   p_id=getpid();          /*process id*/

   p_pid=getppid();        /*parent process id*/

   printf("Process ID: %d\n",p_id);
```

```
    printf("Parent Process ID: %d\n",p_pid);

    system("pstree -p | grep 3221");

    return 0;

}
```

Process Lifecycle

Each process has to go through the following states during it's existence:

1. Creation
2. Running
3. Non-Running
    (a) Ready
    (b) Waiting
4. Termination

Process Creation States

fork() System Call

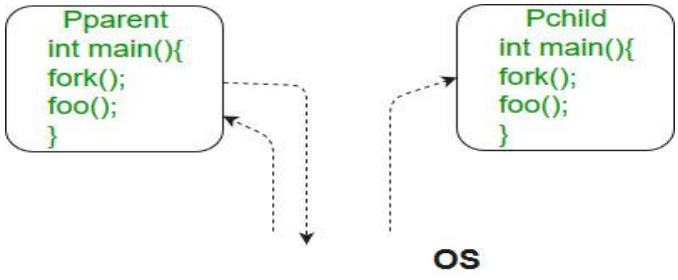
fork system call is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (***parent process***). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc (program counter), same CPU registers, same open files which use in the parent process.

fork system call clones a process into two process.

Fork returns 0 to the child and process ID of the child process to the parent.

It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process. Returns a positive value, the process ID of the child process, to the parent.



1. Predict the Output of the following program

fork1.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        // make two process which run same program after this instruction
        fork();

        printf("Hello world!\n");
        return 0;
}
```

Output:

```
Hello world!

Hello world!
```

2. Predict the Output of the following program

fork2.c

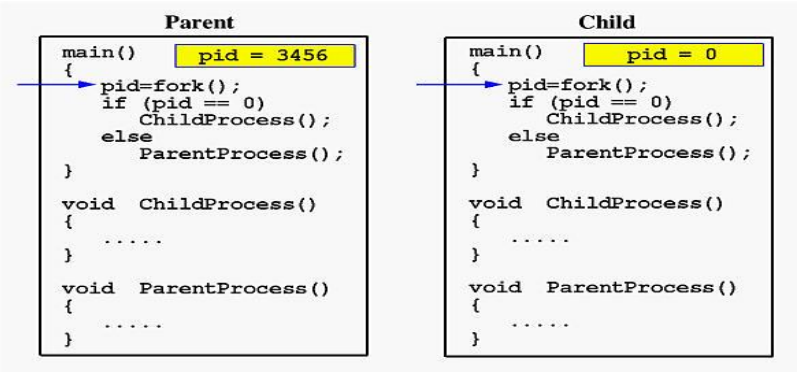
```
// Fork code 2

#include <unistd.h>

#include <stdio.h>      // For printf()

int main()

{

int newly_created;              //  int pid;

newly_created = fork();      //  pid = fork();

if (newly_created == 0)

{
```

```
printf("Child PID = %d\n", newly_created);
printf("I am the Child with PID= %d\n", getpid());
}
else if(newly_created==-1)
{
printf("Error");
}
else
{
printf("Parent of = %d\n", newly_created);
printf("I am the Parent with PPID = %d\n",getppid() );
}
}
```

If pid value is equal to **0** then its shows child, if pid value is **-1** then it shows some error and if pid value is some non-negative and non-zero value then it shows parent process.

newly_created will store the 0 if child process is executed first and store the process id of the newly created child if parent process is executed.

printf("Parent of = %d\n", newly_created); : This statement will show the process id of the newly created child process of the parent process.



3. Calculate number of times hello is printed

fork3.c

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

```
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Output:

```
hello

hello

hello

hello

hello

hello

hello

hello
```

The number of times ‘hello’ is printed is equal to number of process created. Total Number of Processes = 2^n, where n is number of fork system calls. So here n = 3, 2^3 = 8.

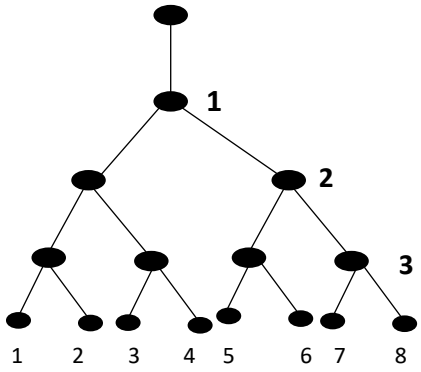
Let us put some label names for the three lines:

```
fork ();   // Line 1

fork ();   // Line 2

fork ();   // Line 3
```

```
       L1       // There will be 1 child process
    /     \     // created by line 1.
  L2      L2    // There will be 2 child processes
 /  \    /  \   //  created by line 2
L3  L3  L3  L3  // There will be 4 child processes
                // created by line 3
```

So, there are total eight processes (new child processes and one original process).

If we want to represent the relationship between the processes as a tree hierarchy it would be the following:



4. **Predict the Output of the following program**

fork4.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    // child process because return value zero
    if (fork() == 0)
            printf("Hello from Child!\n");

    // parent process because return value non-zero.
    else
            printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
```

Output:

1.

```
Hello from Child!
Hello from Parent!
```

(or)

2.

```
Hello from Parent!
Hello from Child!
```

In the above code, a child process is created. fork() returns 0 in the child process and positive integer in the parent process.

Here, two outputs are possible because the parent process and child process are running concurrently. So, we don't know whether the OS will first give control to the parent process or the child process.

Important: Parent process and child process are running the same program, but it does not mean they are identical. OS allocate different data and states for these two processes, and the control flow of these processes can be different.

See next example:

5. **Predict the Output of the following program**

fork5.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d \n", ++x);
    else
        printf("Parent has x = %d \n", --x);
}

int main()
{
    forkexample();
    return 0;
}
```

Output:

```
Parent has x = 0

Child has x = 2

     (or)

Child has x = 2

Parent has x = 0
```

Here, global variable change in one process does not affected two other processes because data/state of two processes are different. And also, parent and child run simultaneously so two outputs are possible.

About process creation concepts, understand and run the following code:

fork6.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
int i;
printf("Process PID %6d \t PPID %6d \n",
getpid(), getppid());

for (i = 0; i<1; ++i)
{
        if (fork()==0)
        {
        printf("Process PID %6d \t PPID %6d \n", getpid(), getppid());
        }
}
return 0;
}
```

Q1: How many processes are created?

Q2: Increase the value in for loop from i<1 to i<2 (i.e., 2 iterations in the loop). Compile and run your program. How many processes does it show this time? Draw a tree hierarchy of processes that you just created as given in Figure- 6.1.

Q3: Increase the value again to i<3 (i.e., 3 iterations). Compile and run your program. How many processes does it show? Draw a tree again. Why is it that we have called fork() 3 times in our code, yet we are seeing 2^n-1 print statement are listed on screen?

Q4: For fun, increase the value yet again to 100. Compile and run. What is going to happen? Does your OS Crash? Does your Program Crash? Can you modify your code to count the total number of fork()s made?

fork7.c

Try and run the following code:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
        fork(); printf("He\n");
        fork(); printf("Ha\n");
        fork(); printf("Ho\n");
}
```

Note that we are calling each He, Ha, and Ho only once. Yet that does not appear when the program is run.

Q 5: Can a Ho be output before a He? Why?

Explanation

The Fork() system call simply creates a new process which is exact replica of parent process.

It requires the unistd.h C library. It's usage is as such:

fork8.c

```
// Fork code 1
#include <unistd.h>
#include <???> // See Question 2
int main()
{
int p;
p = fork();
printf("Job Done\n");      printf("Value of P is %d\n", p);
}
```

To understand this code, try to answer the following questions:

Question. 1: We have used p = fork(). Why not simply fork()? Check **man fork** for answer.

Question.2: Check man page for printf. What library is used for this call?

Question. 3: Run your program. Why is it that printf() is used only once, yet we see the output “Job Done” displaying twice on our screen.

Question.4: Add the following statement to the end of your code and run it again. What output would you see?

printf("Value of P is %d\n", p);

So, the fork() system call will always return different kinds of values (As you should know from Question4). One of the values returned would be “**0**”, the other will be a positive nonzero value.
The process that receives the “0” value is the child process.
The process that receives the positive nonzero value is the parent process.

Based on these values, we can then program our code in such a manner that both parent and child can do their own jobs and not appear to be doing the same things.

Why 0? Why Non-Zero?

The reason for this is that the child can always find out who it's parent is via the getppid() function call. However, it is difficult for the parent to know who it's children are. This can only be done if the value of the child's PID is returned to it at the time of the fork().

Look at the following code structure and implement it:

fork9.c code

```
#include <unistd.h>
#include <stdio.h> // For printf()
int main()
{
int p;
printf("Original Process, pid = %d\n", getpid() );
p = fork();

if (p == 0)
{
printf("Child PID = %d, PPID = %d\n",
getpid(), getppid() );
}
else
{
printf("Parent PID = %d, Child ID = %d\n", getpid(), p);
}
}
```

What would happen if we don't use the If/Else conditions and immediately write the two printf() statements?

Make sure you understand the structure of the Code, as well as what are the ways of knowing the following:

1. The PID
2. The Parent PID
3. The Child PID

Exercise 2

Modify the fork() call code 2 above and add a system call for sleep() after both the printf() statements. Give a time of 120 seconds to the sleep call. While both the parent and child are sleeping, open a new terminal and check out the outcome of **pstree** command as well as the **ps, ps au** command. Study the output for your parent and child process in both commands, especially noting the STAT column.

Exercise 3

Model a fork() call in C/C++ so that your program can create a total of EXACTLY 6 processes (including the parent). (**Note:** You may check the number of processes created using the method from Exercise 1 by using a sleep of 60 seconds or more and entering either ps, or pstree in another terminal).

Your process hierarchy should be as follows:

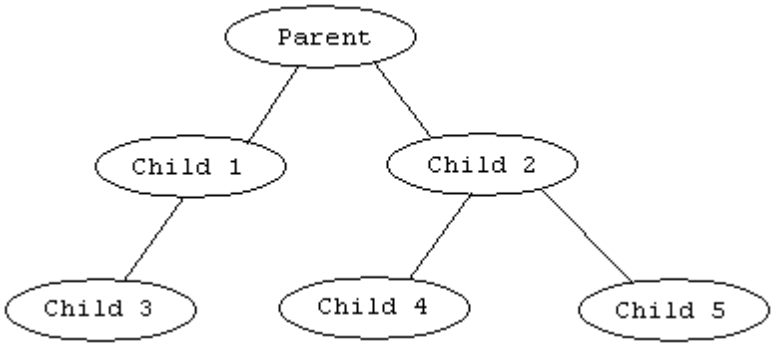


Figure 6.1 Exercise 3 model

```
#include  <stdio.h>
#include  <sys/types.h>
#include  <unistd.h>
#define   MAX_COUNT  10
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()
{
    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()
{
    int   i;
    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n", i);
    printf("*** Parent is done ***\n");
}
```

References

https://www.geeksforgeeks.org/fork-system-call/?ref=leftbar-rightbar

https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html