DHA Suffa University
CS 206 – Operating Systems – Lab
Fall 2017

Lab 08 – Using fork() system call for Process Creation

Twitter: @oslabatdsu

Objective(s):

- Creating process in Linux using fork system call

Process:

- A computer program which is in execution is called process.
- It is a dynamic entity.
- It needs resources-cpu, I/O, allocated when created.

ps Command:

ps command is used to show what processes are currently running on your system.

Process ID:

A Linux or Unix process is running instance of a program. For Example, Firefox is running a process if you are browsing the internet. Each time you start Firefox browser, the system is automatically assigning a unique process identification number (PID). A PID is automatically assigned to each process when it is created. To find out PID of processes, the following commands can be used.

System Calls to get process IDs:

- getpid() - Process ID of child process
- gepppid - Process ID of parent process

```
#include<iostream>
#include<unistd.h>
using namespace std;
int main()
{
    cout << "0. I am process " << getpid() << endl;
    return 0;
}
```

```
                                    osboxes@osboxes ~/Desktop
File  Edit  View  Search  Terminal  Help
osboxes@osboxes ~/Desktop $ g++ fileCode.cpp -o fileCode
osboxes@osboxes ~/Desktop $ ./fileCode
0. I am process 2261
osboxes@osboxes ~/Desktop $
```

Process Creation:

In Unix, process is created by first duplicating the parent process. This is called forking; after the fork system call. In C, you invoke the fork system call with fork() function.

fork()

It is a system call that creates a new process under the Unix Operating system. It takes no arguments. 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().

Example 01 (forking.cpp): using fork() system call

```
#include<iostream>
#include<unistd.h>
using namespace std;
int main()
{
    cout << "Before Forking" << endl;
    fork();
    cout << "After Forking" << endl;
    return 0;
}
```

```
osboxes@osboxes ~/Desktop
File  Edit  View  Search  Terminal  Help
osboxes@osboxes ~/Desktop $ g++ fileCode.cpp -o fileCode
osboxes@osboxes ~/Desktop $ ./fileCode
Before Forking
After Forking
osboxes@osboxes ~/Desktop $ After Forking
```

Example 02 (PidUsage.cpp): using getpid() to get PID

```
#include<iostream>
#include<unistd.h>
using namespace std;
int main()
{
    cout << "0. I am process " << getpid() << endl;
    fork();
    cout << "1. I am process " << getpid() << endl;
    return 0;
}
```

```
osboxes@osboxes ~/Desktop
File  Edit  View  Search  Terminal  Help
osboxes@osboxes ~/Desktop $ g++ fileCode.cpp -o fileCode
osboxes@osboxes ~/Desktop $ ./fileCode
0. I am process 2408
1. I am process 2408
osboxes@osboxes ~/Desktop $ 1. I am process 2409
```

Example 03 (MultipleForking.cpp): Getting the complexity of fork()

```
#include<iostream>
#include<unistd.h>
using namespace std;
int main()
{
    cout << "0. I am process " << getpid() << endl;
    fork();
    cout << "1. I am process " << getpid() << endl;
    fork();
    cout << "2. I am process " << getpid() << endl;
    return 0;
}
```

```
osboxes@osboxes ~/Desktop
File  Edit  View  Search  Terminal  Help
osboxes@osboxes ~/Desktop $ g++ fileCode.cpp -o fileCode
osboxes@osboxes ~/Desktop $ ./fileCode
0. I am process 2547
1. I am process 2547
2. I am process 2547
osboxes@osboxes ~/Desktop $ 2. I am process 2549
1. I am process 2548
2. I am process 2548
2. I am process 2550
```

Some abstract examples of fork() usages:

- Your shell uses fork to run the programs you invoke from the command line.
- Web servers like apache use fork to create multiple server processes, each of which handles requests in its own address space. If one dies or leaks memory, others are unaffected, so it functions as a mechanism for fault tolerance.
- Google Chrome uses fork to handle each page within a separate process. This will prevent client-side code on one page from bringing your whole browser down.
- fork is used to spawn processes in some parallel programs (like those written using MPI). Note this is different from using threads, which don't have their own address space and exist *within* a process.
- Scripting languages use fork indirectly to start child processes. For example, every time you use a command like subprocess.Popen in Python, you fork a child process and read its output. This enables programs to work together.

(Courtesy: tgamblin – StackOverflow Community)

Example 04 (PIDTDataType.cpp): Usage of pid_t datatype

```
#include<iostream>
#include<unistd.h>
using namespace std;
int main()
{
    pid_t childPID;
    childPID = fork();
    for (int i = 0; i < 3; i++)
    {
        cout << "This is process " << getpid() << endl;
        cout << "************************" << endl;
    }
    return 0;
}
```

```
osboxes@osboxes ~/Desktop
File  Edit  View  Search  Terminal  Help
osboxes@osboxes ~/Desktop $ g++ fileCode.cpp -o fileCode
osboxes@osboxes ~/Desktop $ ./fileCode
This is process 2758
************************
This is process 2758
************************
This is process 2758
************************
osboxes@osboxes ~/Desktop $ This is process 2759
************************
This is process 2759
************************
This is process 2759
************************
```

Output of the fork() system call:

- = 0 – Child process successfully created
- > 0 – i.e. the process ID of the child process to the parent process.
- < 0 – Creation of child process was unseuccessful.

The returned process ID is of type pid_t. Normally, the process ID is an integer. Therefore, after the system call to fork(), a simple test can tell which process is the child. 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.

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.

wait (system call)

The parent process may then issue a wait system call, which suspends the execution of the parent process while the child executes. When the child process terminates, it returns an exit status to the operating system, which is then returned to the waiting parent process.

Example 05 (WaitCall.cpp): Working of wait() system call

```
#include<iostream>
#include<unistd.h>      //For fork()
#include<sys/wait.h>    //For wait()
#define COUNT 5
using namespace std;

int  main()
{
     pid_t  pid;
     int    i;

     pid = fork();
     for (i = 1; i <= COUNT; i++)
     {
          cout << "This is from pid: " << pid << " and i is: " << i << endl;
          wait(0);
     }
    return 0;
}
```

```
osboxes@osboxes ~/Desktop
File  Edit  View  Search  Terminal  Help
osboxes@osboxes ~/Desktop $ g++ test.cpp -o test
osboxes@osboxes ~/Desktop $ ./test
This is from pid: 2764 and i is: 1
This is from pid: 0 and i is: 1
This is from pid: 0 and i is: 2
This is from pid: 0 and i is: 3
This is from pid: 0 and i is: 4
This is from pid: 0 and i is: 5
This is from pid: 2764 and i is: 2
This is from pid: 2764 and i is: 3
This is from pid: 2764 and i is: 4
This is from pid: 2764 and i is: 5
osboxes@osboxes ~/Desktop $
```

Lab Task:

a.

```
#include<iostream>
#include<unistd.h>
#include<stdlib.h>
#include<sys/wait.h>
using namespace std;
int main()
{
    if (fork() == 0)
    {
        cout << "1";
    }
    else
    {
        wait(0);
        cout << "0";
        exit(0);
    }
    cout << "0";
    return 0;
}
```

b.

```
#include<iostream>
#include<unistd.h>
#include<stdlib.h>
#include<sys/wait.h>
using namespace std;
int main()
{
    if (fork() == 0)
    {
        cout << "1";
    }
    else
    {
        //wait(0);
        cout << "0";
        exit(0);
    }
    cout << "0";
    return 0;
}
```

What would

- Code ‘a’ print?
- Code ‘b’ print?
- And **WHY?**

Example 06 (GetWait.cpp): Getting the complex working of wait()

```cpp
#include<iostream>
#include<unistd.h>
#include<sys/wait.h>
using namespace std;
int main()
{
    int i, pid;
    pid = getpid();
    cout << "Before fork, pid is " << pid << endl;
    pid = fork();
    cout << "After fork, pid is " << pid << endl;
    if (pid == 0)
    {
        cout << "Child Process starts " << endl;
        for ( int i = 0; i < 3; i++)
        {
            cout << "i is " << i << endl << endl;
        }
        cout << "Child Process ends " << endl;
    }
    else
    {
        cout << "*Wait Starts" << endl;
        cout << "*Process ID: " << getpid() << endl;
        pid = wait (0);
        cout << "*pid: " << pid << endl;
        cout << "*After wait" << endl;
        cout << "*Parent ID: " << getpid() <<endl;
    }
    return 0;
}
```

```
osboxes@osboxes ~/Desktop $ g++ fileCode.cpp -o fileCode
osboxes@osboxes ~/Desktop $ ./fileCode
Before fork, pid is 3697
After fork, pid is 3698
*Wait Starts
*Process ID: 3697
After fork, pid is 0
Child Process starts
i is 0

i is 1

i is 2

Child Process ends
*pid: 3698
*After wait
*Parent ID: 3697
osboxes@osboxes ~/Desktop $
```

Lab Assignment 08

1. **Complete the following programs. How many output lines are printed? Check the source code and submit its source code file as well as a document (pdf) containing screenshots of the output.**

a.

```
void fn()
{
    fork();
    fork();
    cout << "Hello" << endl;
    return;
}
int main()
{
    fn();
    cout << "Hello" << endl;
    exit(0);
}
```

b.

```
void fn()
{
    if( fork() == 0)
    {
        fork();
        cout << "Hello" << endl;
        exit(0);
    }
    return;
}
int main()
{
    fn();
    cout << "Hello" << endl;
    exit(0);
}
```

c.

```
void fn()
{
    if( fork() == 0)
    {
        fork();
        cout << "Hello"  << endl;
        return;
    }
    return;
}
int main()
{
    fn();
    cout << "Hello" << endl;
    exit(0);
}
```

d.

```
int counter = 1;
int main()
{
    if (fork() == 0)
    {
        counter--;
        exit(0);
    }
    else
    {
        wait(NULL);
        counter++;
        cout << "counter = " << counter << endl;
    }
    exit(0);
}
```

Submission Instructions:

1. **Number your .sh files as question number e.g. Q1a.cpp, Q1a.pdf, Q1b.cpp, Q1b.pdf, etc. (Q is in upper case)**
2. **Create a new folder named cs152abc where abc is your 3 digit roll #. e.g. cs152111.**
3. **Copy all the .cpp and pdf files into this folder.**
4. **Right-click on the folder you created and create a zip file by selecting the option**
   - **"Send to" and selecting "Compressed (zipped) folder" [for windows].**
   - **"Create Archive" and change option to ".zip" instead of ".tar.gz" and click on "Create". [for linux]**

   Now make sure a zip file named cs152abc.zip is created e.g. *cs152111.zip*.
5. **Upload the assignment solution on LMS under the assignment named Lab 08 Assignment – XX, where XX is your section name.**
6. **Due Date for assignment submission is Oct 22, 2017.**