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

Lab 10 – UNIX Process Control

Objective(s):

- Creating process in Linux using fork system call

exec() system call:

The exec() system call is used after a fork() system call by one of the two processes to replace the memory space with a new program. The exec() system call loads a binary file into memory (destroying image of the program containing the exec() system call) and go their separate ways. Within the exec family there are functions that vary slightly in their capabilities.

exec family:

1. execl() and execlp():

execl(): It permits us to pass a list of command line arguments to the program to be executed. The list of arguments is terminated by NULL. e.g.

int execl(const char *_path_, const char *_arg_, ...);

Example:

```
execl("/bin/ls", "ls", "-l", NULL);
```

execlp(): It does same job except that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used. The function execlp() can also take the fully qualified name as it also resolves explicitly. e.g.

int execlp(const char *_file_, const char *_arg_, .)

Example:

```
execlp("ls", "ls", "-l", NULL);
```

Example 01 (ExeclORExeclp.cpp): Working of execl and execlp system call

```
#include<iostream>
#include<unistd.h>
using namespace std;
int main()
{
    pid_t pid = fork();
    if ( pid == 0 )
    {
        cout << "Child Process" << endl;
        execl("/bin/dir", "dir", NULL); //Executing dir in bash
                // OR
        execlp("dir", "dir", NULL); //Executing dir in bash
    }
    else if ( pid > 0 )
    {
        cout << "Parent Process " << endl;
    }
    else
    {
        cout << "Error" << endl;
    }
    return 0;
}
```

2. execv() and execvp():

execv(): It does same job as execl() except that command line arguments can be passed to it in the form of an array of pointers to string. e.g.

int execv(const char *_path_, char *const _argv_[]);

Example

```
char *argv[] = ("ls", "-l" ,NULL);
execv("/bin/ls", argv);
```

execvp(): It does same job expect that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used. e.g.

int execvp(const char *_file_, char *const _argv_[]);

Example

```
char *argv[] = ("ls", "-l" ,NULL);
 execvp("ls", argv);
```

Example 02 (ExecvOORExecvp.cpp): Working of execv and execvp system call

```
#include<iostream>
#include<unistd.h>
using namespace std;
int main()
{
    pid_t pid = fork();
    char *cmd [] = {"dir", NULL};
    if ( pid == 0 )
    {
        cout << "Child Process" << endl;
        execv("/bin/dir", cmd); //Executing dir in bash
              // OR
        execvp("dir", cmd); //Executing dir in bash
    }
    else if ( pid > 0 )
    {
        cout << "Parent Process " << endl;
    }
    else
    {
        cout << "Error" << endl;
    }
    return 0;
}
```

Example 03 (Exec.cpp): Working of exec with wait analyzing variables' behavior

```cpp
#include<iostream>
#include<unistd.h>
#include<sys/types.h>
#include<stdlib.h>
#include<sys/wait.h>
using namespace std;
int global = 0;
int main()
{
    pid_t childPID;
    int status;
    int local;
    childPID = fork();
    if ( childPID >= 0 )
    {
        if ( childPID == 0)
        {
            cout << endl << endl << "Child Process! " << endl;
            local++;
            global++;
            cout << "Child PID = " << getpid() << " Parent PID = " << getppid() << endl;
            cout << "Child's local = " << local << " Child's global = " << global << endl;
            char* cmd []= {"ps", NULL};
            execv("/bin/ps", cmd);
        }
        else
        {
            cout << endl << endl <<"Parent Process!" << endl;
            cout << "Parent PID = " << getpid() << " CHild PID = " << childPID << endl;
            wait(&status);
            cout << "Child Exit Code = " << WEXITSTATUS(status) << endl;
            cout << "Parent's local = " << local << " and Parent's global = " << global << endl;
            cout << "Parent says Bye!!!!" << endl;
            exit(0);
        }
    }
    else
    {
        cout << "Error Forking" << endl;
        exit(0);
    }
    return 0;
}
```

Zombie Process:

To understand what a zombie process is and what causes zombie processes to appear, you'll need to understand a bit about how processes work on Linux.

When a process dies on Linux, it isn't all removed from memory immediately — its process descriptor stays in memory (the process descriptor only takes a tiny amount of memory). The process's status becomes EXIT_ZOMBIE and the process's parent is notified that its child process has died with the SIGCHLD signal. The parent process is then supposed to execute the wait() system call to read the dead process's exit status and other information. This allows the parent process to get information from the dead process. After wait() is called, the zombie process is completely removed from memory.

This normally happens very quickly, so you won't see zombie processes accumulating on your system. However, if a parent process isn't programmed properly and never calls wait(), its zombie children will stick around in memory until they're cleaned up.

How do I see if there are zombie processes on a system?

Run "ps aux" and look for a Z in the STAT column.

Example 04 (Zombie.cpp): Zombie Process

```
#include<iostream>
#include<unistd.h>
#include<stdlib.h>
using namespace std;
int main()
{
    int pid = getpid();
    cout << "I am Process: " << pid << endl;
    cout << "[Forking child Process...]" << endl;
    pid = fork();
    if ( pid < 0 )
    {
        exit(-1);
    }
    else if (pid == 0)
    {
        cout << "Child Process Started." << endl;
        cout << "Child Process Completed" << endl;
    }
    else
    {
        usleep(5000000);
        cout << "Parent Process Running" << endl;
        cout << "I am in Zombie State" << endl;
        while (1)
        {
        }
    }
    return 0;
}
```

Orphan Process:

An orphan process is a computer process whose parent process has finished or terminated, though itself remains running. A process may also be intentionally orphaned so that it becomes detached from the user's session and left running in the background; usually to allow a long-running job to complete without further user attention, or to start an indefinitely running service. Under UNIX, the latter kinds of processes are typically called daemon processes. The UNIX *nohup* command is one means to accomplish this.

Example 05 (Orphan.cpp): Orphan Process

```cpp
#include<iostream>
#include<unistd.h>
using namespace std;
int main()
{
    pid_t p = fork();
    if ( p == 0 )
    {
        sleep(5);
    }
    cout << "The Child Process pid is: " << getpid() << " Its Parent pid: " << getppid() << endl;
    sleep(10);
    cout << endl << "Process " << getpid() << " is done. Its Parent pid is: " << getppid() << endl;
    return 0;
}
```

Example 06 (OrphanStates.cpp): Orphan state of a process

```cpp
#include<iostream>
#include<unistd.h>
#include<stdlib.h>
using namespace std;
int main()
{
    int pid = getpid();
    cout << "Current Process ID is: " << pid << endl;
    cout << "[Forking Child Process...]" << endl;
    pid = fork();
    if (pid < 0)
    {
        exit(-1);
    }
    else if ( pid == 0)
    {
        cout << "Child Process is Sleeping..." << endl;
        sleep(5);
        cout << "Orphan Child's Parent ID: " << getppid() << endl;
    }
    else
    {
        cout << "Parent Process Completed " << endl;
    }
    return 0;
}
```

Lab Assignment 09:

1. Create a C++ program that forks the process(es) in such a way that one of the processes (if many) is shown both as a Zombie and Child Process. Use exec() family's system call(s) in your cpp file to print the zombie state of the process and show pid on the terminal to confirm that the same process is also an orphan.

Submission Instructions:

1. **Create a new folder named cs152abc where abc is your 3 digit roll #. e.g. cs152111.**
2. **Copy all the solution files into this folder.**
3. **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]**
4. **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 Nov 05, 2017.**