# Lab 3 – PROCESSES

##### OBJECTIVE

Learn to create processes using fork() system call. **TIME REQUIRED** : 3 hrs **PROGRAMMING LANGUAGE** : C/C++

**SOFTWARE REQUIRED** : Ubuntu/Fedora, gcc/gc, Text Editor, Terminal, Windows, Dev

**HARDWARE REQUIRED** : Core i5 in Computer Labs

##### PROCESS CREATION

A process is a program in execution. The process which creates another process is called parent process. The process which is created is called child process. We can identify process by their unique key called process identifier or pid (integer number). In Linux we can use fork() system call to create processes. By this system call new process is created containing copy of parent process. Both process (parent and child) continue executing instructions after fork(). The return number of fork() for new (child) process will be 0 (zero), whereas for parent process value will be nonzero positive process identifier. If fork() fails, it return a negative number. In this section, we create a simple program using fork() to create child process using

##### TASK 3.1

**Using** getpid(): This function returns the pid of the current program. Use the following code and write the output.

int main(){

int pid;

pid = getpid();

printf(“Process ID is %d\n”, pid);

return 0;

}

##### TASK 3.2

What is the outcome of the following program?

int main(){

long i;

printf(“Process ID is %d\n”, getpid());

for(i=0; i<=400;i++){

printf(“i is %d\n”, i);

}

return 0;

}

##### TASK 3.3

**Using** getppid(): This function returns the pid of the parent process.

int main(){

int ppid;

ppid = getppid();

printf(“Parent Process ID is %d\n”, ppid);

return 0;

}

What is the outcome?

##### TASK 3.4

**Using** fork(): fork command in Linux creates a new process by duplicating the calling process. The new process, referred to as the **child**, is an exact duplicate of the calling process, referred to as the **parent**. What is the outcome of the following program?

int main(){

fork();

printf(“The PID is %d\n”, getpid());

return 0;

}

##### TASK 3.5

What is the outcome of the following program?

int main(){

int pid;

pid = fork(); if(pid==0){

printf(“I am child, my process ID is %d\n”, getpid()); printf(“The parent process ID is %d\n”, getppid());

}

else{

printf(“I am parent, my process ID is %d\n”, getpid()); printf(“The parent process ID is %d\n”, getppid());

}

return 0;

}

##### TASK 3.6

To see if the pid is same as shown in the system, Open System Monitor. Check to see if the pid is same. Use the following code

int main(){

int pid,i; pid = fork(); if(pid==0){

printf(“I am child, my process ID is %d\n”, getpid()); printf(“The parent process ID is %d\n”, getppid());

}

else{

printf(“I am parent, my process ID is %d\n”, getpid()); printf(“The parent process ID is %d\n”, getppid());

}

scanf(“%d”,&i); //so that program halts for user input return 0;

}

Show screenshots here: -

##### TASK 3.7

What is the outcome of this program?

/\*\*

* This program forks a separate process using the fork()/exec() system calls.
* \* Figure 3.10\*
* @author Gagne, Galvin, Silberschatz Operating System Concepts - Seventh Edition
* Copyright John Wiley & Sons - 2005. \*/ #include <stdio.h>

#include <unistd.h> #include <sys/types.h> int main(){

pid\_t pid;

/\* fork a child process \*/ pid = fork();

if (pid < 0) { /\* error occurred \*/ fprintf(stderr, "Fork Failed\n"); exit(-1);

}

else if (pid == 0) { /\* child process \*/ printf("I am the child %d\n",pid); execlp("/bin/ls","ls",NULL);

}

else { /\* parent process \*/

/\* parent will wait for the child to complete \*/ printf("I am the parent %d\n",pid);

wait(NULL);

printf("Child Complete\n"); exit(0);

}

}

##### EXERCISE 3.1 [2]

Explain how was the execution carried out in program in last code? You may use a flow model to describe it.

##### TASK 3.8

**Using** Exec**:** The *fork* system call creates a new process but that process contains, and is executing, exactly the same code that the parent process has. More often than not, we'd like to run a new program.

**Example of Execve():**The *execve* system call replaces the current process with a new program. Type the following command in Terminal and show the output.

ls -aF /

Now execute the following code

/\* execve: run a program \*/

#include <stdlib.h> /\* needed to define exit() \*/ #include <unistd.h> /\* needed to define getpid() \*/ #include <stdio.h> /\* needed for printf() \*/

int main(int argc, char \*\*argv) {

char \*args[] = {"ls", "-aF", "/", 0};

/\* each element represents a command line argument \*/ char \*env[] = { 0 };

/\* leave the environment list null \*/

printf("About to run /bin/ls\n"); execve("/bin/ls", args, env);

perror("execve"); /\* if we get here, execve failed \*/ exit(1);

}

What is the outcome?

##### TASK 3.9

**Using** Execlp

*execlp*, which allows you to specify all the arguments as parameters to the function. Note that the first parameter is the command. The second parameter is the first argument in the argument list that is passed to the program (argv[0]). These are often the same but don't have to be. The last parameter must be a null pointer.

/\* execlp: run a program using execlp \*/

#include <stdlib.h> /\* needed to define exit() \*/ #include <unistd.h> /\* needed to define getpid() \*/ #include <stdio.h> /\* needed for printf() \*/

int main(int argc, char \*\*argv) { printf("About to run ls\n");

execlp("ls", "ls", "-aF", "/", (char\*)0); perror("execlp"); /\* if we get here, execlp failed \*/ exit(1);

}

What is the output?

##### TASK 3.10

**Using** fork() and exec(): The fork system call creates a new process. The execve system call overwrites a process with a new program. A process forks itself and the child process execs a new program, which overlays the one in the current process.

/\* forkexec: create a new process. \*/

/\* The child runs "ls -aF /". The parent wakes up after 5 seconds

\*/

#include <stdlib.h> /\* needed to define exit() \*/ #include <unistd.h> /\* needed for fork() and getpid() \*/ #include <stdio.h> /\* needed for printf() \*/

int main(int argc, char \*\*argv) { void runit(void);

int pid; /\* process ID \*/ switch (pid = fork()) {

case 0: /\* a fork returns 0 to the child \*/ runit();

break;

default: /\* a fork returns a pid to the parent \*/ sleep(5); /\* sleep for 5 seconds \*/ printf("I'm still here!\n");

break;

case -1: /\* something went wrong \*/ perror("fork");

exit(1);

}

exit(0);

}

void runit(void) {

printf("About to run ls\n");

execlp("ls", "ls", "-aF", "/", (char\*)0); perror("execlp"); /\* if we get here, execlp failed \*/ exit(1);

}

What is the outcome of the program?

##### EXERCISE 3.2 [2]

What do you understand from last code?

##### TASK 3.11

##### Creating a Process in Windows

/\*\*

* This program creates a separate process using the CreateProcess() system call. Figure 3.12
* @author Gagne, Galvin, Silberschatz Operating System Concepts - Seventh Edition
* Copyright John Wiley & Sons - 2005.

\*/

#include <windows.h>

#include <stdio.h>

#include <tchar.h>

int main( int argc, TCHAR \*argv[] )

{

STARTUPINFO si;

PROCESS\_INFORMATION pi;

ZeroMemory( &si, sizeof(si) );

si.cb = sizeof(si);

ZeroMemory( &pi, sizeof(pi) );

if( argc != 2 )

{

printf("Usage: %s [cmdline]\n", argv[0]);

return 1;

}

// Start the child process.

if( !CreateProcess( NULL, // No module name (use command line)

argv[1], // Command line

NULL, // Process handle not inheritable

NULL, // Thread handle not inheritable

FALSE, // Set handle inheritance to FALSE

0, // No creation flags

NULL, // Use parent's environment block

NULL, // Use parent's starting directory

&si, // Pointer to STARTUPINFO structure

&pi ) // Pointer to PROCESS\_INFORMATION structure

)

{

printf( "CreateProcess failed (%d).\n", GetLastError() );

return 1;

}

// Wait until child process exits.

WaitForSingleObject( pi.hProcess, INFINITE );

// Close process and thread handles.

CloseHandle( pi.hProcess );

CloseHandle( pi.hThread );

}

What is the Outcome?

##### EXERCISE 3.3 [6]

Which OS gives you a better interface for creating and executing child programs and why?

##### RESOURCES

<http://manpages.ubuntu.com/manpages/lucid/man2/fork.2.html> <http://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/exec.html> <http://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/forkexec.html> <http://www.advancedlinuxprogramming.com/alp-folder/alp-ch03-processes.pdf>