CSC3150 Project1 Report

118010045 Cui Yuncong

October 2020

Contents

1 Task1

1.1 Design

Since Task 1 is about making another child process and execute a file within child process, I had to use fork() first. Then in child process, first use raise() to raise the signal, then use execve() to execute a test program. While child process is executing a file, parent process waits until child process terminates. In this case, parent process use waitpid() to wait and receive a signal from child process. After child process terminates and parent process receives a signal from child process, based on the type of signal, parent process prints what kind of signal it receives.

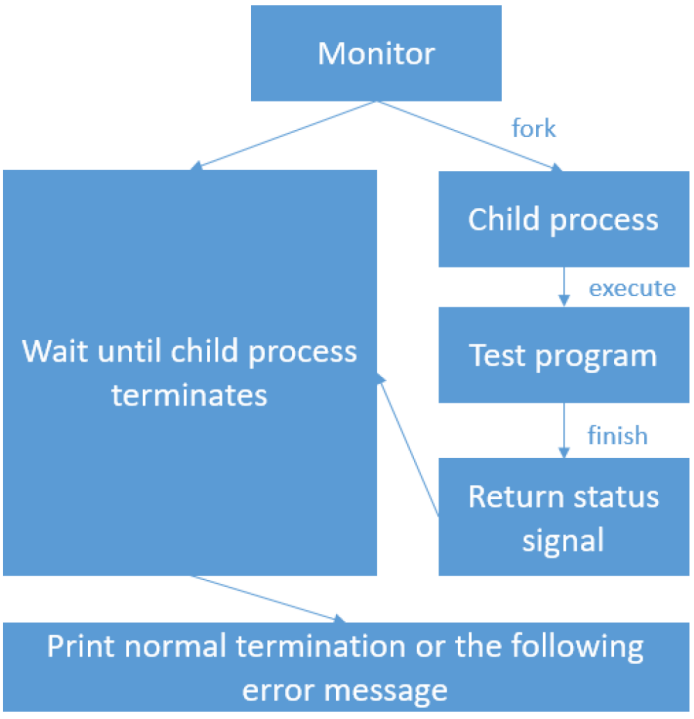


Figure 1: Main Flow Chart

1.2 Running Environment

Version of OS: Ubuntu 16.04.2

Kernel: Linux 4.10.14

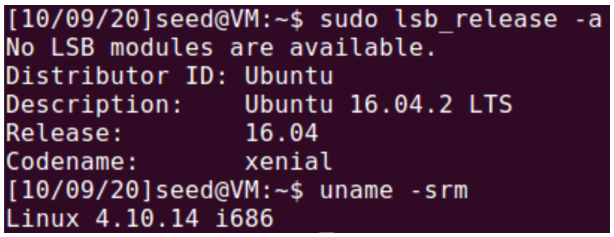


Figure 2: Version of OS and Kernel

1.3 Execution

In terminal, type "find . -exec touch {} \;" to touch every file in order to avoid clock skew detected problem just in case (Not Necessary).

Type "make" to compile all the files such as main and tests files.

Type "./program1 ./FILENAME" to execute specific test file.

Inside the program, program will first fork the process. Child process execute a test file while parent process waits for child process to terminate. Based on signal received from child process, parent process prints out the result.

1.4 Program Output

1.4.1 Normal Termination

```
[10/09/20]seed@VM:~/.../program1$ ./program1 ./normal
Process start to fork
I'm the parent process, my pid = 8662
I'm the child process, my pid = 8663
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the normal program

------------CHILD PROCESS END------------
Parent process receving the SIGCHLD signal
Normal termination with EXIT STATUS = 0
```

Figure 3: Normal Termination

1.4.2 Signaled Abort

```
[10/09/20]seed@VM:~/.../program1$ ./program1 ./abort
Process start to fork
I'm the parent process, my pid = 8707
I'm the child process, my pid = 8708
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGABRT program

Parent process receving the SIGCHLD signal
child process get SIGABRT signal
child process is abort by abort signal
CHILD EXECUTION FAILED!!
```

Figure 4: Signaled Abort

1.4.3 Stopped

```
[10/09/20]seed@VM:~/.../program1$ ./program1 ./stop
Process start to fork
I'm the parent process, my pid = 8671
I'm the child process, my pid = 8672
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receving the SIGCHLD signal
child process get SIGSTOP signal
child process stopped
CHILD PROCESS STOPPED
```

Figure 5: Stopped

1.5 Learning Remarks

1. Using the fork() function in the program can fork a child process. The function will return the parent process pid(process identifier) for the parent process, return 0 for the child process and return -1 if the fork failed. The variable type for the pid is pid_t. After the child process being forked, the system will return an identical program in child process. Both child process and parent process will continue to execute the line after the fork function. The getpid() function will return the pid of the current process and the getppid() function will return the pid of the parent of current process.

2. There are multiple functions for executing an external program in a child process, which are following They are of the same function but with different input parameters. After execvp() function, the child process will start to execute the external program from the very beginning. Any code in the child process after the execvp() function will not be executed. The function will not return anything if the execution function succeeds. It will return -1 if fails.

3. Since the terminated child process without parent's wait will become a zombie and the terminated child process with parent's termination will become an orphan, it is important to make sure that parent process will wait for the child process to terminate with waitpid() function. The wait function only requires the status of the child process while the waitpid function need three input parameters which are pid, status and option. The waitpid function has three options which are 0, WNOHANG and WUNTRACED. 0 skips the option, WNOHANG requires the signal information immediately and WUNTRACED option will

wait for the child process to terminate and return the signal information.

4. There are 64 signals in linux and each of them have a corresponding signal code. The code and the signal can be shown by typing "kill -l" in the terminal. The signals are divided into three categories which are exited, signaled and stopped. The three categories of signals can be detected by WIFEXITED, WIFSIGNALED, WIFSTOPPED separately. The WIFEXITED signal will return if the process exited smoothly, the system will usually ignore such signals. The WIFSIGNALED signal will return if the process is terminated. The WIFSTOPPED signal will return if the process is stopped. The function needs the status of the child process as the input and output the integer value. If the signal belongs to the corresponding category, the function will return non-zero, otherwise it will return 0. In order to find the specific signal code for each signal. We can also use WEXITSTATUS, WTERMSIG, WSTOPSIG for each category.

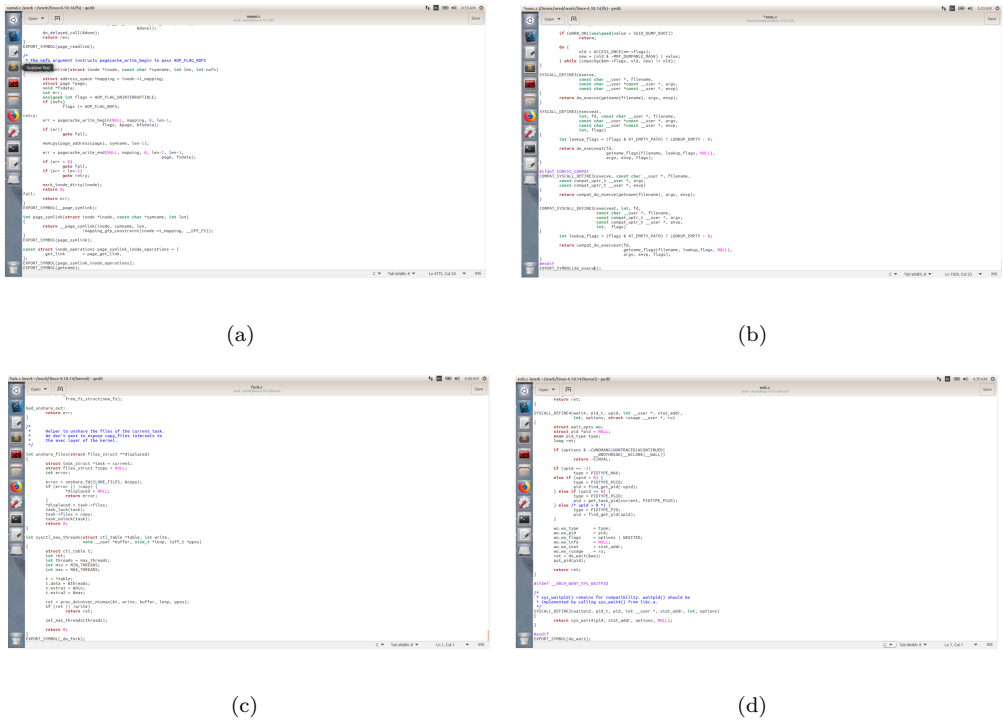
```
[10/10/20]seed@VM:~/.../bonus$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Figure 6: 64 Signals in Linux

2 Task2

2.1 Design

In Task 2, it needs to initialize Kernel module first, then I need to create kernel thread. An important thing in this task is that before I use kernel related function, I need to go to the folder that includes all the kernel information like header files, then I need to find the specific header file that include the function that I want to use. After that, use EXPORT_SYMBOL for that function. The last is re-compiling kernel. Inside prgrogram2.c, I also need to use extern to use the function I exported. _do_fork(), do_execve(), and do_wait() work same as Task1, except I need to put additional parameter for each function.



(a) (b)

(c) (d)

Figure 7: The Modification of Kernel Files (a): namei.c (b): exec.c (c): fork.c (d): exit.c

Inside the program, it will initialize kernel module and create kernel thread. After that, it will fork a process. In child process, it will execute a test program. Parent process waits until child process terminates and prints the signal number based on signal it received from child process. It will print module_exit if we call rmmod in terminal.

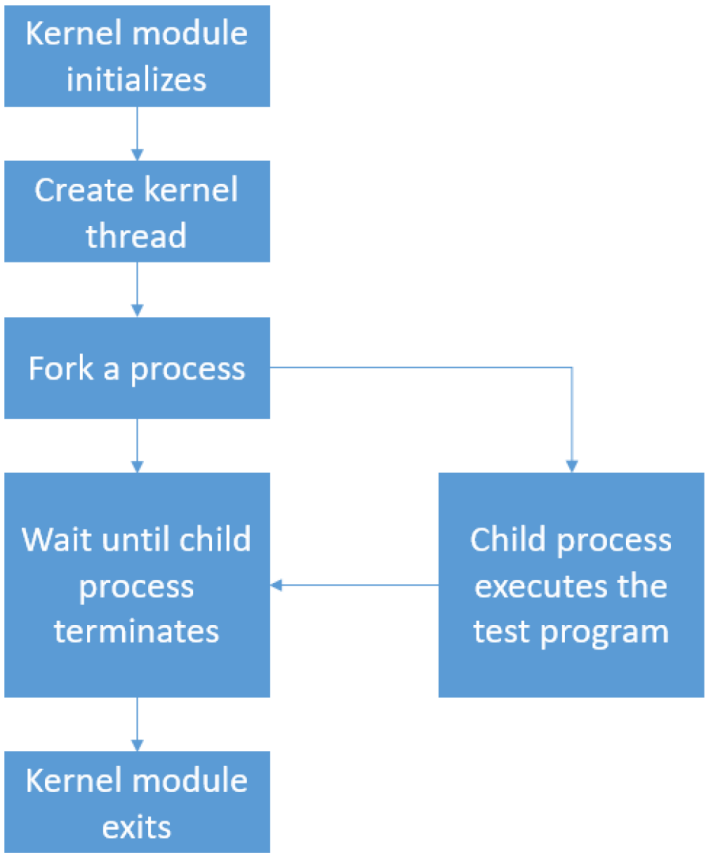


Figure 8: Main Flow Chart

2.2 Running Environment

Version of OS: Ubuntu 16.04.2

Kernel: Linux 4.10.14

```
[10/09/20]seed@VM:~$ sudo lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.04.2 LTS
Release:        16.04
Codename:       xenial
[10/09/20]seed@VM:~$ uname -srm
Linux 4.10.14 i686
```

Figure 9: Version of OS and Kernel

2.3 Execution

Type “sudo su” to get the root authority first.

In terminal, type “find . -exec touch {} \;” to touch every file in order to avoid clock skew detected problem just in case (Not Necessary).

Type “gcc test.c -o test” to compile test.c.

Type “make” to make module, which is program2.ko.

Type “insmod program2.ko” to insert module.

Type “rmmod program2” to remove module.

Type “dmesg” to check the message buffer in kernel.

```
[10/11/20]seed@VM:~/.../program2$ gcc test.c -o test
[10/11/20]seed@VM:~/.../program2$ make
make -C /lib/modules/4.10.14/build M=/home/seed/Desktop/source/program2 modules
make[1]: Entering directory '/home/seed/work/linux-4.10.14'
  CC [M]  /home/seed/Desktop/source/program2/program2.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seed/Desktop/source/program2/program2.mod.o
  LD [M]  /home/seed/Desktop/source/program2/program2.ko
make[1]: Leaving directory '/home/seed/work/linux-4.10.14'
[10/11/20]seed@VM:~/.../program2$ sudo insmod program2.ko
[10/11/20]seed@VM:~/.../program2$ sudo rmmod program2
[10/11/20]seed@VM:~/.../program2$ dmesg
```

Figure 10: Execution Steps

2.4 Program Output

```
[10/11/20]seed@VM:~/.../program2$ dmesg | grep program2
[  218.934324] [program2] : module_init
[  218.934771] [program2] : module_init create kthread start
[  218.934772] [program2] : module_init kernel thread start
[  218.934850] [program2] : The child process has pid = 5015
[  218.934851] [program2] : This is the parent process, pid = 5013
[  218.935299] [program2] : get SIGBUS signal
[  218.935299] [program2] : child process has SIGBUS error
[  218.935300] [program2] : The return signal is 7
[  233.161285] [program2] : module_exit
```

Figure 11: Test Result

2.5 Learning Remarks

1. We use do_fork function load a child process in the kernel module. After the function executed, it is loading the kernel fork.ko module.

2. We use the do_execute or _do_execute function to let the child process execute the external program in kernel mode. It will return the non-zero value if the execution succeeds otherwise it will return 0. The child process will not execute any lines in the original file after the execution function.

3. The parent process waits for the child process to terminate in the kernel mode, which needs the struct wait_opts and do_wait function to let the parent process wait. Wait_opts is a struct which contains multiple of process information. It also needs to decrease the count and free the memory after do_wait function.

4. To handle the signal in kernel model, we can use the *wo.wo_stat or the status form the wait_opts to determine the return signal.

5. Remove the kernel module from the kernel in order to execute the exit function

3 Bonus Question

3.1 Design

This program is written to implement the functions which is able read multiple test programs and recursively fork the child process. The program is able to return the pid of each process including the main process of "myfork.c" file and is also able to return the signal raised in each input file. We implement the program with the idea of recursion and store every signal and pid.

1. We first read every input file and put the test files into a character array (exclude the myfork file). The integer variable $argc$ is the number of the input file.

2. Then the main function will recur with the initial index of 1 and totally input number of $argc - 1$ and the file list arg.

3. Whenever the recursion is called, it will fork a child process with the vfork() function since we need the parent and the child process to share the variables. As long as the process is not for the last input file, the child process will continuously recur.

4. The parent process will catch the signal and pid of their child process. It also need to execute the input files in order to pass signal and pid information to their parent unless it does not have the parent process. When we reached the process for the last input file. We will execute the last file in the input file.

5. After the recursion, we print the process tree. The first pid value in the process tree should be the value of current process(myfork). After that we will print the rest pid value we have stored before. We continuously print the rest information of each child process.

3.2 Running Environment

Version of OS: Ubuntu 16.04.2

Kernel: Linux 4.10.14

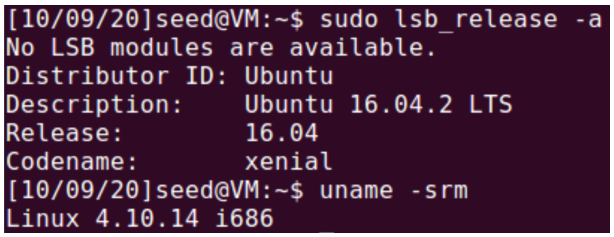


Figure 12: Version of OS and Kernel

3.3 Execution

In terminal, type "find . -exec touch {} \;" to touch every file in order to avoid clock skew detected problem just in case (Not Necessary).

Type "make" to compile all the files such as main and tests files.

Type "./myfork filename1 filename2 filename3" to execute specific test file (the filenameX is the name of the external test file).

The user should input multiple test external file (at least one file and at most 45 files) and the output will be the pid of each process and the signals raisedin the files.

3.4 Program Output

```
[10/10/20]seed@VM:~/.../bonus$ ./myfork hangup normal3 trap
------------CHILD PROCESS START------------
This is the SIGTRAP program

This is normal3 program
------------CHILD PROCESS START------------
This is the SIGHUP program

The process tree: 4208->4209->4210->4211
The child process (pid=4211) of parent process (pid=4210) is terminated by signal
Its signal number is 5
child process got SIGTRAP signal
child was terminated by trap signal

The child process (pid=4210) of parent process (pid=4209) has normal execution
Its exit status = 0

The child process (pid=4209) of parent process (pid=4208) is terminated by signal
Its signal number is 1
child process got SIGHUP signal
child was terminated by hang up signal


Myfork process(pid=4208) execute normally
```

Figure 13: Myfork with hangup, normal3 and trap

3.5 Learning Remarks

1. Using vfork() function to fork a child process can let the processes share the variables. With the vfork() function, the program will always execute the child process before the parent process.

2. Using the recursion to continuously fork the child process and execute the external file.

3. Manipulate multiple processes.