CSC3150 Operating Systems

Project 1 report

Name: Xiao Nan

Student ID: 119010344

Date: 2021/10/08

The Chinese University of Hong Kong, Shen Zhen

0. Introduction:

In task 1 I wrote a program which can fork a child process executing a test program and catch the signal returned from the child process in order to analyze it and print out the signal information. In task 2 I constructed a kernel module, which can fork a child process in kernel space. The child process can execute a test program during initialization and return results to the original program. The results are printed in the kernel log. In bonus task by using Shared Memory I implemented a program which can print out a process tree and the test program termination information when multiple test programs are passed in as arguments.

1. Design of the program:

In program 1 I plan to first initiate the variables which can store the process ID (PID) and state information. Then I fork a child process using fork() API and store either the child process ID (in parent process) or the child process identified (0 for the child process). Then I used this variable to separate the behaviors of the parent process and that of the child process by judging whether the variable has value 0. In the child process the test program is executed using execve() API. In the parent process the termination status of the child process is accepted by waitpid() function and the macro corresponding to the status code is printed out.

In program 2 I defined the functions of my_exec() and my_wait() to handle the waiting and function executing requirements of the program. During the initialization of the module, the program creates a kernel thread using

kthread_create() and within the thread, a process is waken up using wake_up_process(), which will further initialize the default signal handlers for the parent process and then fork a child process using the kernel function _do_fork() and execute test programs. In the child process, I implemented the my_exec() function to specify execution arguments and execute the test program. I wrote the my_wait() function to initialize waiting options (wait_opts) and to invoke do_wait() and to catch the exit code of the child process. Then in the parent process after receiving the exit code, the program would formulate the code by masking the first 5 bits of the exit code (& 0b11111) to transfer the code into the corresponding signal's code. Finally, the caught signal and its information are printed out. During the exit of the module the information for exiting would be printed out.

In the bonus problem I implemented a program which can execute programs in the order given by the command line arguments as well as print out the process tree and the program execution information. I used two arrays-one storing locally (corresponding to only a certain parent-child process pair) the returning status of the child process and the other storing locally the PID of the parent and child process-for the convenience of keeping track of the information (PID and returning status) relative to each pair of parent-child processes. I also used two shared arrays (allocated using mmap()) to store the PID and returning status of all processes in order that I can print all of them out when child processes terminate. The main part of the execution of the program lies in a "while" loop, where the first parent process would fork an initial child process. Then each of

the other child processes would fork another child process, except for the last child process, which only updates the PID information to the shared array as well as executes the last program. Then the parent processes of the last child process, which have called waitpid() to wait for the execution of the child process, would receive the execution status, store them in the shared array of statuses, and execute the program passed in the second to the last argument. After the parent process of the last child process finished the execution, the parent process acts as the child process of the second last parent process, which again stores the status to the shared array and execute the third last program. Other parent processes do the same things as the second last parent process except for the first parent process, which just breaks the "while" loop and prints out the resulting status and parent-children PID's information.

2. Runtime environment:

2.1. Linux version:

```
1. namshoo@ubuntu:/$ cat /etc/issue
2. Ubuntu 16.04.5 LTS \n \l
```

2.2. Linux kernel version for problem 1 and bonus problem:

```
1. namshoo@ubuntu:/$ uname -r
2. 4.15.0
```

2.3. Linux kernel version for problem 2:

```
1. namshoo@ubuntu:/$ uname -r
2. 4.10.14
```

2.4 GCC version:

```
1. namshoo@ubuntu:/$ gcc --version
```

```
2. gcc (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609
3. Copyright (C) 2015 Free Software Foundation, Inc.
4. This is free software; see the source for copying conditions. T
   here is NO
5. warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICU
   LAR PURPOSE.
```

3. Steps to execute my program:

In command line, you need to first unzip the .zip file. Make sure that the .zip file is under the current directory:

```
1. namshoo@ubuntu:~$ unzip Assignment_1_119010344.zip
```

3.1 Problem 1:

"cd" to the program1 execution directory, and type "make" to compile the source codes and the test programs. Make sure that the Makefile is under the current directory:

```
1. namshoo@ubuntu:~$ cd Assignment_1_119010344/source/program1
2. namshoo@ubuntu:~/Assignment_1_119010344/source/program1$ make
```

Then, type "./program1 ./test_program" to implement the test case.

```
1. namshoo@ubuntu:~/Assignment_1_119010344/source/program1$ ./pro
   gram1 ./stop # The second argument can be the name of
   any test program you like
```

3.2 Problem 2:

"cd" to the program2 execution directory and type "make" to compile the kernel module object files and the test case:

```
1. namshoo@ubuntu:~$ cd Assignment_1_119010344/source/program2
2. namshoo@ubuntu:~/Assignment_1_119010344/source/program2$ make
```

Enter the root user mode and insert the module. Check if the module had been successfully inserted:

```
namshoo@ubuntu:~/Assignment_1_119010344/source/program2$ sudo su
[sudo] password for namshoo:
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# insmod program2.ko
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# lsmod | grep program2
program2               2958  0
```

After the signal had been received by the initialization process and the initialization process had completed, remove the module from the kernel, and then check the kernel log for the output information:

```
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# rmmod program2.ko
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# dmesg | tail -n 20
```

3.3 Bonus problem:

"cd" to the bonus program execution directory and type "make" to compile the source codes and the test cases:

```
namshoo@ubuntu:~$ cd Assignment_1_119010344/source/bonus
namshoo@ubuntu:~/Assignment_1_119010344/source/bonus$ make
```

Then type "./myfork ./test_program1 ./test_program2……" to execute the program. The number of test programs following the "./myfork" argument can be arbitrary:

```
namshoo@ubuntu:~/Assignment_1_119010344/source/bonus$ ./myfork ./kill ./alarm ./normal1 ./pipe ./normal9 # The number of test programs can be arbitrary. Just make sure the test program object files exist in the current directory
```

4. Program output screenshot:

4.1 Program 1:

Normal test program output is shown in figure 4.1. The stopped and failure test program outputs are shown in figure 4.2 as well as 4.3:

```
namshoo@ubuntu:~/Assignment_1_119010344/source/program1$ ./program1 ./normal
Process start to fork
I'm the parent process, my pid = 7537
I'm the child process, my pid = 7538
Child process starts to execute the test procram:
------------CHILD PROCESS START------------
This is the normal program

------------CHILD PROCESS END------------
Parent process receives SIGCHLD signal
Normal exit
Normal termination with EXIT STATUS = 0
namshoo@ubuntu:~/Assignment_1_119010344/source/program1$
```

Figure 4.1: Normal output

```
namshoo@ubuntu:~/Assignment_1_119010344/source/program1$ ./program1 ./alarm
Process start to fork
I'm the parent process, my pid = 7554
I'm the child process, my pid = 7555
Child process starts to execute the test procram:
------------CHILD PROCESS START------------
This is the SIGALRM program

Parent process receives SIGCHLD signal
Child process get SIGALRM signal
Child process gets the alarm signal from timer
CHILD EXECUTION FAILED: 14
namshoo@ubuntu:~/Assignment_1_119010344/source/program1$
```

Figure 4.2: Failure output

```
namshoo@ubuntu:~/Assignment_1_119010344/source/program1$ ./program1 ./stop
Process start to fork
I'm the parent process, my pid = 7552
I'm the child process, my pid = 7553
Child process starts to execute the test procram:
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receives SIGCHLD signal
Child process get SIGSTOP signal
Child process is stopped
CHILD PROCESS STOPPED: 19
namshoo@ubuntu:~/Assignment_1_119010344/source/program1$
```

Figure 4.3: Stopping output

4.2 Program 2:

Normal test program output is shown in figure 4.4. The outputs of SIGBUS signal and stopped programs are shown in figure 4.5, 4.6, respectively:

```
namshoo@ubuntu:~/Assignment_1_119010344/source/program2$ sudo su
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# insmod program2.ko
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# dmesg | tail -n 8
[10128.912363] [program2] : Module_init
[10128.912364] module_init create kthread start
[10128.912444] Kthread starts
[10128.912897] The parent process is inited with pid: 9579
[10128.912906] The child process id is: 9581
[10128.912906] Child process starts
[10128.921769] normal exit
[10128.921770] child process terminated
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# rmmod program2.ko
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# dmesg | tail -n 11
[10003.129277] [program2] : Module_exit
[10128.912363] [program2] : Module_init
[10128.912364] module_init create kthread start
[10128.912444] Kthread starts
[10128.912897] The parent process is inited with pid: 9579
[10128.912906] The child process id is: 9581
[10128.912906] Child process starts
[10128.921769] normal exit
[10128.921770] child process terminated
[10128.921771] The return signal is: 0
[10135.742484] [program2] : Module_exit
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2#
```

Figure 4.4: Normal output

```
namshoo@ubuntu:~/Assignment_1_119010344/source/program2$ sudo su
[sudo] password for namshoo:
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# insmod program2.ko
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# dmesg | tail -n 8
[ 9847.586922] [program2] : Module_init
[ 9847.586923] module_init create kthread start
[ 9847.587033] Kthread starts
[ 9847.587454] The parent process is inited with pid: 8708
[ 9847.587463] The child process id is: 8710
[ 9847.587498] Child process starts
[ 9847.711307] get SIGBUS signal
[ 9847.711308] child process terminated
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# rmmod program2.ko
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# dmesg | tail -n 11
[ 9722.274065] [program2] : Module_exit
[ 9847.586922] [program2] : Module_init
[ 9847.586923] module_init create kthread start
[ 9847.587033] Kthread starts
[ 9847.587454] The parent process is inited with pid: 8708
[ 9847.587463] The child process id is: 8710
[ 9847.587498] Child process starts
[ 9847.711307] get SIGBUS signal
[ 9847.711308] child process terminated
[ 9847.711308] The return signal is: 7
[ 9857.388156] [program2] : Module_exit
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2#
```

Figure 4.5: SIGBUS output

```
namshoo@ubuntu:~/Assignment_1_119010344/source/program2$ sudo su
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# insmod program2.ko
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# dmesg | tail -n 8
[ 9990.699004] [program2] : Module_init
[ 9990.699005] module_init create kthread start
[ 9990.699322] Kthread starts
[ 9990.700015] The parent process is inited with pid: 9144
[ 9990.700028] The child process id is: 9146
[ 9990.700028] Child process starts
[ 9990.713792] get SIGSTOP signal
[ 9990.713793] child process terminated
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# rmmod program2.ko
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2# dmesg | tail -n 11
[ 9982.555803] [program2] : Module_exit
[ 9990.699004] [program2] : Module_init
[ 9990.699005] module_init create kthread start
[ 9990.699322] Kthread starts
[ 9990.700015] The parent process is inited with pid: 9144
[ 9990.700028] The child process id is: 9146
[ 9990.700028] Child process starts
[ 9990.713792] get SIGSTOP signal
[ 9990.713793] child process terminated
[ 9990.713794] The return signal is: 19
[10003.129277] [program2] : Module_exit
root@ubuntu:/home/namshoo/Assignment_1_119010344/source/program2#
```

Figure 4.6: Stopped output

4.3 Bonus program:

The result of executing 10 test programs is shown in figure 4.7:

```
namshoo@ubuntu:~/Assignment_1_119010344/source/bonus$ ./myfork ./normal5 ./normal10 ./pipe ./terminate ./alarm ./segment_fault
------------CHILD PROCESS START------------
This is the SIGSEGV program

------------CHILD PROCESS START------------
This is the SIGALRM program

------------CHILD PROCESS START------------
This is the SIGTERM program

------------CHILD PROCESS START------------
This is the SIGPIPE program

This is normal10 program
This is normal5 program
Process tree: 9759->9760->9761->9762->9763->9764->9765
Child process 9765 of parent process 9764 is terminated by signal 11 (SIGSEGV)
Child process 9764 of parent process 9763 is terminated by signal 14 (SIGALAM)
Child process 9763 of parent process 9762 is terminated by signal 15 (SIGTERM)
Child process 9762 of parent process 9761 is terminated by signal 13 (SIGPIPE)
Child process 9761 of parent process 9760 is terminated by signal 0 (normal exit)
Child process 9760 of parent process 9759 is terminated by signal 0 (normal exit)
Myfork process (9759) terminated normally
namshoo@ubuntu:~/Assignment_1_119010344/source/bonus$
```

Figure 4.7: Output of executing 10 test programs

5. Things I've learnt from the project:

I learnt a lot of useful techniques dealing with kernel-mode programming and system call invocation. In program 1 I understood the mechanism how a parent process can fork a child process and how the parent process can catch signals from the

child process. In program 2 I understood how a kernel mode is initialized and how a kernel thread is created. I also realized that many user APIs invoking the system call ultimately rely on kernel functions (like _do_fork(), do_wait()) to finish their tasks, and that the kernel provides many useful data structures (for example, the task_struct for process descriptor and sighand_struct for signal handling and actions management) for process management. In program 3 I understood the shared memory, a powerful tool for Inter-Process Communication. I also learnt using mmap() and munmap() to allocate or deallocate the shared memory for practical use.