CSC3050 Operating System

Project Report #1

Name: Xu Xiangyu

Student ID: 120090446

Date: 2022.10.9

1. Environment

a) Linux Version

The Linux version is ubuntu Ubuntu 16.04.7.

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# cat /etc/issue
Ubuntu 16.04.7 LTS \n \l
```

b) Linux Kernel Version

The Linux kernel version is 5.10.146.

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# uname -r
5.10.146
```

c) GCC Version

The GCC version is 5.4.0.

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

2. Task 1

a) Design

There are four sections to this work. Forking a child process in user mode is the first step. Following that, the child process must run the test program, and the parent process must wait for the child process to end. Last but not least, the termination state should be verified by identifying the returned status signal.

i. Fork a child process

Create a child process by using the method fork(). A number will be returned by this function and kept in the pid variable. If there is a mistake, pid will be negative. In the parent process pid is the pid ID of the child process while in the child process it is zero.

ii. Child process executes the test program

The test programs will be run in this section by child processes. There are 15 available test programs. The name of the currently running program should be kept in *arg[argc]. Therefore, it can be run in a child process by entering the program's name.

iii. Parent process waits for child process termination

Using waitpid, the parent process can wait for the child process to finish (). The flag for the third parameter should be WUNTRACED so that the process can also return when a child process has stopped.

iv. Check child process's termination status

There are three different types of termination status: normal termination, failing case, and stopped. These three different types of status signals are distinguished using the functions WIFEXITED(), WIFSIGNALED(), and WIFSTOPPED(). There are other 13 exceptions that the child process failed. It is possible to get the signal number by using the WTERMSIG() method.

b) Execution

i. Compile: Enter the "make" command in the "program1" directory.

ii. Execute: In the 'program1' directory, type './program1 $TEST_CASE $ARG1 $ARG2 ...', where $TEST_CASE is the name of test program and $ARG1, $ARG2,... are names of arguments that the test program could have. Then you can see the output of the test program.

c) Output

Below are some examples of test programs.

i. Demo output for normal termination

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./normal
Process start to fork
I'm the Parent Process, my pid = 7425
I'm the Child Process, my pid = 7426
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the normal program

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

ii. Demo output for trapping

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./trap
Process start to fork
I'm the Parent Process, my pid = 8223
I'm the Child Process, my pid = 8224
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGTRAP program

Parent process receives SIGCHLD signal
Child process was trapped
SIGTRAP signal was raised in child process
```

iii. Demo output for terminated process

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./terminate
Process start to fork
I'm the Parent Process, my pid = 8146
I'm the Child Process, my pid = 8147
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGTERM program

Parent process receives SIGCHLD signal
Child process terminated
SIGTERM signal was raised in child process
```

iv. Demo output for segment fault

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./segment_fault
Process start to fork
I'm the Parent Process, my pid = 4664
I'm the Child Process, my pid = 4665
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGSEGV program

Parent process receives SIGCHLD signal
Child process had segment fault
SIGSEGV signal was raised in child process
```

v. Demo output for quiting

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./quit
Process start to fork
I'm the Parent Process, my pid = 4586
I'm the Child Process, my pid = 4587
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGQUIT program

Parent process receives SIGCHLD signal
Child process quited
SIGQUIT signal was raised in child process
```

vi. Demo output for broken pipe

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./pipe
Process start to fork
I'm the Child Process, my pid = 4311
I'm the Parent Process, my pid = 4310
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGPIPE program

Parent process receives SIGCHLD signal
Child process had broken pipe
SIGPIPE signal was raised in child process
```

vii. Demo output for kill

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./kill
Process start to fork
I'm the Parent Process, my pid = 4162
I'm the Child Process, my pid = 4163
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGKILL program

Parent process receives SIGCHLD signal
Child process was killed
SIGKILL signal was raised in child process
```

viii. Demo output for interruption

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./interrupt
Process start to fork
I'm the Parent Process, my pid = 4067
I'm the Child Process, my pid = 4068
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGINT program

Parent process receives SIGCHLD signal
Child process was interrupted
SIGINT signal was raised in child process
```

ix. Demo output for illegal instruction

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./illegal_instr
Process start to fork
I'm the Parent Process, my pid = 3840
I'm the Child Process, my pid = 3841
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGILL program

Parent process receives SIGCHLD signal
Child process had illegal instruction
SIGILL signal was raised in child process
```

x. Demo output for hang up

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./hangup
Process start to fork
I'm the Parent Process, my pid = 3763
I'm the Child Process, my pid = 3764
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGHUP program

Parent process receives SIGCHLD signal
Child process was hung up
SIGHUP signal was raised in child process
```

xi. Demo output for floating

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./floating
Process start to fork
I'm the Parent Process, my pid = 3685
I'm the Child Process, my pid = 3686
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGFPE program

Parent process receives SIGCHLD signal
Child process had floating point exception
SIGFPE signal was raised in child process
```

xii. Demo output for bus

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./bus
Process start to fork
I'm the Parent Process, my pid = 3626
I'm the Child Process, my pid = 3627
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGBUS program

Parent process receives SIGCHLD signal
Child process had bus error
SIGBUS signal was raised in child process
```

xiii. Demo output for alarm

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./alarm
Process start to fork
I'm the Parent Process, my pid = 3548
I'm the Child Process, my pid = 3549
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGALRM program

Parent process receives SIGCHLD signal
Child process was alarmed
SIGALRM signal was raised in child process
```

xiv. Demo output for abort

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./abort
Process start to fork
I'm the Child Process, my pid = 3466
I'm the Parent Process, my pid = 3465
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGABRT program

Parent process receives SIGCHLD signal
Child process was aborted by abort signal
SIGABRT signal was raised in child process
```

xv. Demo output for stopped

```
root@csc3150:/home/vagrant/csc3150/120090446/program1# ./program1 ./stop
Process start to fork
I'm the Child Process, my pid = 8100
Child process start to execute test program:
I'm the Parent Process, my pid = 8099
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receives SIGCHLD signal
child process get SIGSTOP signal
```

3. Task 2

a) Design

There are six sections to this task. First part is to update the Linux source code. Then a kernel thread is created to implement the my_fork() function. The rest parts are similar to that of the first task , except they are carried out in kernel mode.

i. Update the Linux source code

There are a few non-static functions used in this program's module. The functions can be used in another module if they are exported using EXPORT SYMBOL(). The kernel module is rebuilt and installed after the source code has been altered. The new boot image is used in place of the original.

ii. Create a kernel thread

A kernel thread is constructed by using kernel_create(), and the thread starts through wake_up_process().

iii. Fork a child process

Fork function is implemented by the function my_fork(). The kernel uses the default sigaction setting for current process to keep track of which signals are currently pending or muted as well as how each thread group is expected to handle each signal. To fork a child process, use the kernel_clone() function.We need to struct a kernel_clone_args structure and give it to kernel_clone() function. The stack field needs to be (unsigned long) &my_exec while exit_signal needs to be SIGCHLD. &my_exec will implement the execution of a test program in the child process. Before using this function, it needs to be externalized in the program file.

iv. Child process executes the test program

To implement the execution of a test program in a child process, the function my_exec() was developed. The test program is run using the path using the do_execve() function from the kernel module. When the system call do_execve() is used in kernel mode, the exec.ko module is loaded. In '/fs/exec.c', its implementation is specified. Before using this function, it needs to be externalized in the program file.

v. Parent process waits for child process termination

The parent process's wait function is implemented by the method my_wait(). The parent process is made to wait for the child process to finish by using the do_wait() method from the kernel module. When the system call do_wait() is used in kernel mode, the exit.ko module is loaded. '/kernel/exit.c' contains a definition of its implementation. Additionally, a structure called wait opts that will be utilized in the implementation of the do_wait() function needs to be made.

vi. Check child process's termination status

This section is similar to the related section in task 1. To detect different status signals, however, since the functions WIFEXITED(), WIFSIGNALED(), WIFSTOPPED(), and WTERMSIG() cannot be used, the structure's parameter wo.wo_stat should be used. It is an int, which represents the child process's termination state. When the value is 0, it means that the process has ended normally.

b) Execution

i. Compile the test program: In the 'program2' directory, type 'gcc -o test test.c'.

ii. Compile the main program: In the 'program2' directory, type 'make' command and enter.

iii. Debug

1. Type 'sudo insmod program2.ko' under 'program2' directory and enter.
2. Type 'sudo rmmod program2' and enter to remove the program2 module.
3. See messages appear by typing 'dmesg -c' command.

iv. Try Different test file: To test other termination signals, you can alter the signal raised in test.c or swap out the test programs in task1 for the path in my exec(). Then, repeat the procedures above.

c) Output

i. Demo output for normal termination

```
root@csc3150:/home/vagrant/csc3150/120090446/program2# dmesg -c
[11988.036879] [program2] : Module_init Xu Xiangyu 120090446
[11988.064648] [program2] : Module_init create kthread start
[11988.112804] [program2] : Module_init kthread start
[11988.145716] [program2] : The child process has pid = 10555
[11988.162420] [program2] : This is the parent process, pid = 10554
[11988.185584] [program2] : child process
[11988.188903] [program2] : child process normal terminated with EXIT STATUS = 0
[11989.977003] [program2] : Module_exit
```

ii. Demo output for bus

```
root@csc3150:/home/vagrant/csc3150/120090446/program2# dmesg -c
[11548.503385] [program2] : Module_init Xu Xiangyu 120090446
[11548.503388] [program2] : Module_init create kthread start
[11548.503636] [program2] : Module_init kthread start
[11548.503792] [program2] : The child process has pid = 8948
[11548.503793] [program2] : This is the parent process, pid = 8947
[11548.503794] [program2] : child process
[11548.599138] [program2] : get SIGBUS signal
[11548.599139] [program2] : child process had bus error
[11548.599140] [program2] : The return signal is 7
[11550.526614] [program2] : Module_exit
```

iii. Demo output for terminated

```
root@csc3150:/home/vagrant/csc3150/120090446/program2# dmesg -c
[12082.514853] [program2] : Module_init Xu Xiangyu 120090446
[12082.560724] [program2] : Module_init create kthread start
[12082.617655] [program2] : Module_init kthread start
[12082.645667] [program2] : The child process has pid = 11023
[12082.649608] [program2] : This is the parent process, pid = 11022
[12082.694145] [program2] : child process
[12082.708691] [program2] : get SIGTERM signal
[12082.741812] [program2] : child process terminated
[12082.764802] [program2] : The return signal is 15
[12085.798174] [program2] : Module_exit
```

iv. Demo output for stop

```
root@csc3150:/home/vagrant/csc3150/120090446/program2# dmesg -c
[16260.321331] [program2] : Module_init Xu Xiangyu 120090446
[16260.372689] [program2] : Module_init create kthread start
[16260.422344] [program2] : Module_init kthread start
[16260.467971] [program2] : The child process has pid = 12354
[16260.480213] [program2] : This is the parent process, pid = 12353
[16260.487632] [program2] : child process
[16260.492726] [program2] : get SIGSTOP signal
[16260.495282] [program2] : child process stopped
[16263.107395] [program2] : Module_exit
```

4. Environment Set Up

a) Set up VM: The first step is Install vitrualbox and vagrant and set up ubuntu.

b) Set up VS Code: The second step is set up Remote SSH plugin in VS Code so we can use ubuntu remotely by VS Code.

c) Change kernel version to 5.10.146 and compile kernel

i. Download source code from http://www.kernel.org.

ii. Install Dependency and development tools.

iii. Extract the source file to /home/seed/work.

iv. Copy config from /boot to /home/seed/work/KERNEL_FILE.

v. Login root account and go to kernel source directory.

vi. Clean previous setting and start configuration.

vii. Build kernel Image and modules.

viii. Install kernel modules use $make modules_install.

ix. Install kernel use $make install.

x. Reboot to load new kernel.

xi. Check exiting kernel version to see if we succeed.

5. Conclusion

a) Gain from this assignment

i. The environment set up for ubuntu coding use VS Code.

ii. Some Linux command basic knowledges.

iii. Kernel installing and compiling.

iv. The creation of processes in both user mode and kernel mode, as well as the connections between parent and child processes.

v. Kernel module thread creation.

vi. Process signals raising and receiving.

vii. Kernel source code modifying.

b) Feedback to this assignment

This assignment is an extension and good practice of codes in tutorial. I developed a stronger understanding of concepts like process and thread as I improved the code and dealt with numerous errors frequently. Instead than needing to be corrected over and over by piazza, it would be wonderful if the guidelines for the upcoming task were more explicit.