CSC3150
Assignment1

120090358 Yuantao Zhang

October 10, 2022

1 Overview

This assignment mainly contains two parts except the bonus part. Task one requires us to fork a child process and run the test program in user mode. After the child process terminates (finishes or raises some signals), it sends the SIGCHLD signal to parent process. Parent process should output some information according to received signals. Task two requires us to modify the kernel code to fork a child process and run the test program. Because the real implementation of fork() and execve() (these two functions are called in user mode, which are used in task one) will call the codes in kernel, task two wants us to read the kernel source code and utilize some functions to implement them.

2 My design

In task one, at the very beginning, a 'pid' variable is created. Then I use the fork() function to fork a child porcess. To distinguish the child process and parent process, the 'pid' variable is used(both parent process and child process run the same codes, but have different 'pid' value). If the value of the 'pid' variable is 0, then the current process is the child process. After retriving the paramaters from the input, the child process begins to run the test program and will send SIGCHLD signal back to parent process after terminating.

If the value of the 'pid' variable is not zero, then the current process is the parent process. Parent process will wait child process to terminate and receives the signals from child process. According to different signals reveived, different information will be output. The 'switch-case' language is used to control the output flow of parent process.

In task two, I download the kernel source code first and comile the kernel. After compilation, I install the kernel to the virtual machine. Then I start to write my code. In my program, there are three main functions, $my\_exec()$, $my\_fork()$, and $program2\_init()$. when the program of task two starts to run, a kernel thread will be created by the $program2\_init()$ fucntion to run $my\_fork()$, which is the parent process. The $my\_fork()$ function first sets the default sigaction for current process. Then the $kernel\_clone()$ function is called to fork a child process which executes the $my\_execve()$ function. After that, the $do\_wait()$ function will receive the signal from the child process and output relevant information. In $my\_execve()$, the child process first uses $getname\_kernel()$ to get the filename of the test program according to the absolute path. The result of $getname\_kernel()$ is passed as the parameter of $do\_execve()$ to execute the test program.

3 Development environment Setting

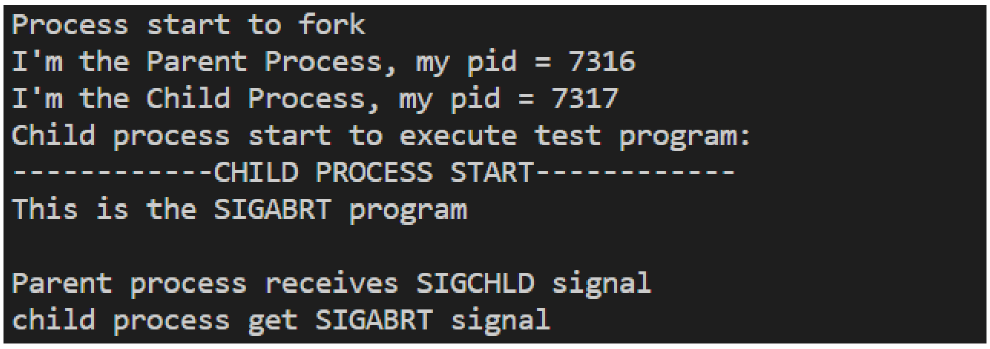
The development environment of the whole assignment is quite crucial. There are two problems of the development environment of this assignment. First is that the requirement of the kernel version is 5.10.x, so the kernel source code needs to be downloaded from official website, compiled and installed. Second is that task two needs to use some functions in the kernal source code, because of the default convenience setting, several symbols of functions and varaibles in kernel source code are invisible to the users. Thus, if users want to use those symbols, the $EXPORT\_SYMBOL()$ function should be called to export those symbols from the kernel source code and make those symbols visible to users. Given that the $EXPORT\_SYMBOL()$ function should be written in the source code, the kernal needs to be recompiled after the modification.

To solve the first difficulty, I download the kernal source code of version 5.10.147 and decompress the downloaded file. After that, I installed some packages on the VM and set up the .config file to prepare for the kernel compilation and installation. Then, I use the "make BZImage" and "make Module" instructions to compile and install the kernal.

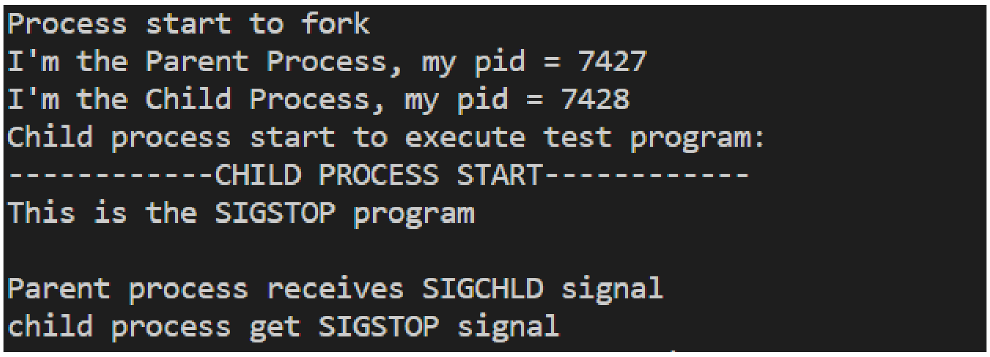
To solve the second difficulty, I find the position of the source code of *do_wait*(), *getname_kernel*(), *do_execve*(), and *kernel_clone*(). Behind the function declaration of these functions in the source code, I add the *EXPORT_SYMBOL*() instruction to all of them. Then I recompile the kernel use the "make BZImage" and "make Module" instructions again.

4 Running output

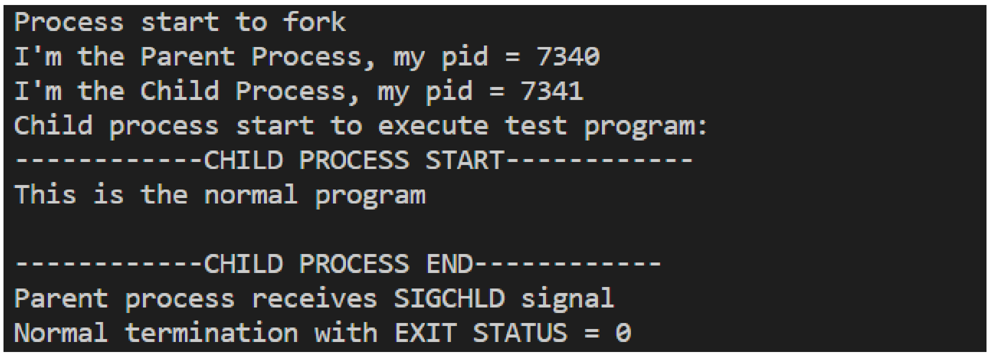
In this part, I will demonstrate the result of my codes of task one and task two(take abort, stop ,normal as examples).
Task one:



```
Process start to fork
I'm the Parent Process, my pid = 7316
I'm the Child Process, my pid = 7317
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGABRT program

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

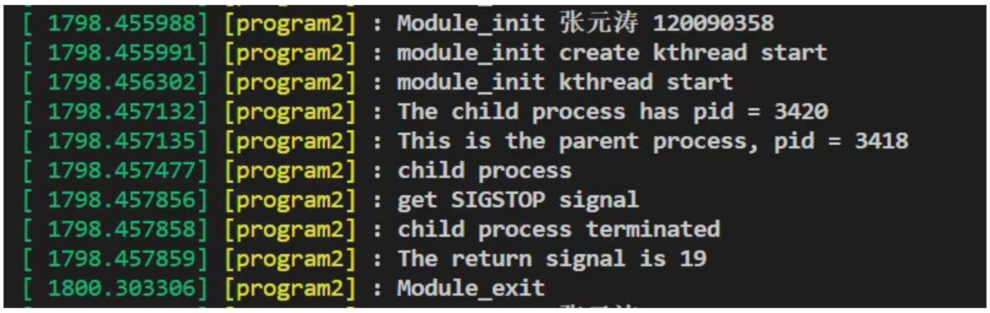


```
Process start to fork
I'm the Parent Process, my pid = 7427
I'm the Child Process, my pid = 7428
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGSTOP program

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

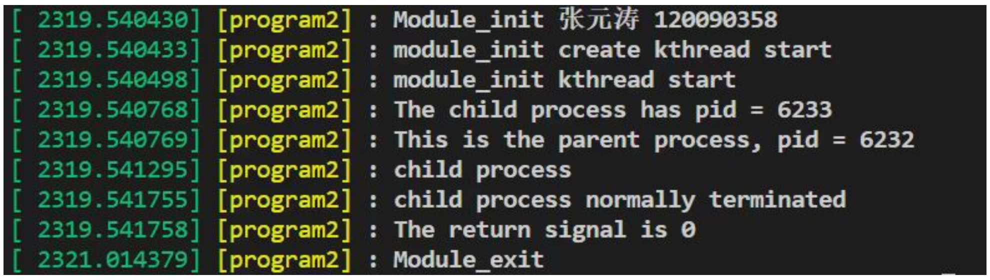


```
Process start to fork
I'm the Parent Process, my pid = 7340
I'm the Child Process, my pid = 7341
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the normal program

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

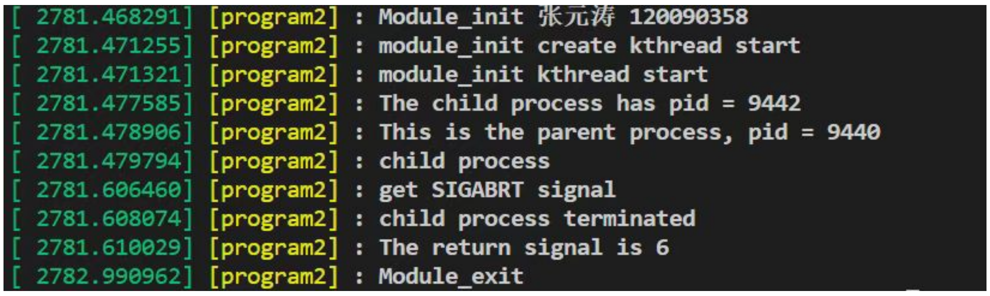
Task two:



```
[ 1798.455988] [program2] : Module_init 张元涛 120090358
[ 1798.455991] [program2] : module_init create kthread start
[ 1798.456302] [program2] : module_init kthread start
[ 1798.457132] [program2] : The child process has pid = 3420
[ 1798.457135] [program2] : This is the parent process, pid = 3418
[ 1798.457477] [program2] : child process
[ 1798.457856] [program2] : get SIGSTOP signal
[ 1798.457858] [program2] : child process terminated
[ 1798.457859] [program2] : The return signal is 19
[ 1800.303306] [program2] : Module_exit
```



```
[ 2319.540430] [program2] : Module_init 张元涛 120090358
[ 2319.540433] [program2] : module_init create kthread start
[ 2319.540498] [program2] : module_init kthread start
[ 2319.540768] [program2] : The child process has pid = 6233
[ 2319.540769] [program2] : This is the parent process, pid = 6232
[ 2319.541295] [program2] : child process
[ 2319.541755] [program2] : child process normally terminated
[ 2319.541758] [program2] : The return signal is 0
[ 2321.014379] [program2] : Module_exit
```



```
[ 2781.468291] [program2] : Module_init 张元涛 120090358
[ 2781.471255] [program2] : module_init create kthread start
[ 2781.471321] [program2] : module_init kthread start
[ 2781.477585] [program2] : The child process has pid = 9442
[ 2781.478906] [program2] : This is the parent process, pid = 9440
[ 2781.479794] [program2] : child process
[ 2781.606460] [program2] : get SIGABRT signal
[ 2781.608074] [program2] : child process terminated
[ 2781.610029] [program2] : The return signal is 6
[ 2782.990962] [program2] : Module_exit
```

5 What I get from this project

Actually, this project brings me a deeper insight into the kernel and the linux OS. I learned to know thousands of instructions of linux. For example, I learned to use ”chmod” to change the file permission of files; I learned to use ”dmesg” to see the kernel log. Besides, after doing this project, I think the OS kernel is very close to us. The source code of kernel is like normal program and we can use the way to understand our normal codes to understand the kernel source code. Fianlly, I have some feelings of characteristics of the OS kernel. The complilation time of kernel source codes is very long, whcih means that the kernel basically involves almost all functions of operating system. Given that the definition of kernel is the codes that always run on the computer, we can simply insert the codes into kernel and see the result of the kernel log to test our programs.