Kernel Mode Programming

Yohandi [SID: 120040025]

Program Design

Program 1 The program begins with a fork procedure. This procedure is performed by invoking the fork() function that returns an integer value when being invoked. If the returned value is -1, it indicates that the forking process has failed. Aside from that, it indicates that the forking process is successful. As a result, we will have two operating processes. One of the processes obtains a return value of zero, which is known as the child process. The other process, namely the parent process, is a process that returns a value equal to the process identification number of the child process. The process of forking permits the children to allocate a separate heap outside of the heap of the parent process. This heap contains nearly all of the content of the parent process. Using this knowledge, we can directly use the inputs from the parent process during program execution to execute the required program in the child process. This is possible with the execve() method. The new program will not return any value upon successful execution. If it fails, the code will return to the child process code in the parent program. The parent process is then prepared to catch any signal sent by the child process. In order to accomplish this, we implement the use of waitpid() function, which is designed to detect and wait for any signal raised by the child process. After receiving any signal, it assigns the return value of the child process to one of its allocated arguments. Using this specified argument, we can then examine the execution status of the child process and inform its users.

Program 2 To begin, the program creates a kernel thread to execute the forking procedure. The forking procedure is contained within the my_fork() method. Then, within this method, we do the forking operation by using the function kernel_clone(). In addition, we send the my_exec() function pointer to the kernel_clone() function. This will be the function that contains all the code for child processes. In contrast to program 1, the parent and child processes' codes are separated into two distinct functions. If the function kernel_clone() is successfully executed, it will return the PID of the child process. In addition, the child process will begin to execute the code within the my_exec() function. In the code of child processes, external file execution must be performed manually. In other words, the path of the desired program must be specified directly in the code of the child process and not as one of the execution arguments of the parent process program. After supplying the path in the code of the child process, the do_execve() function can be used to execute the desired program. Similar to program 1, the program will only return to the code of the child process in the event of a mistake. By implementing the use of the function do_wait() in the parent process, we will wait for the return value of the child process. After configuring the wait function, the return value is analyzed. A return result of 0 indicates a normal termination status. The termination possibilities are involved when the return value is smaller than 256. Other than that, the program is considered to receive the stop signal. After processing the signal, the program notifies the user and returns to the main menu.

Program Execution

Program Environtment

- Linux Kernel Version: Linux-5.10.146
- GCC Version: gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)

Set up the Development Environtment We first install both Virtual Machine (from the virtualbox official website) and Vagrant (from the vagrantup official website). Then, we launch PowerShell as an administrator to up a vagrant for the CSC3150 folder with the main server, cyzhu/csc3150. Then, we set up the ssh remote in Visual Studio Code by installing the Remote - SSH extension and config it with the SSG-TARGETS. After copying all the necessary information, we will be able to open a new window named default. Last, we open a new terminal and install all essential dependencies and libraries by typing `sudo apt update && sudo apt install -y build-essential`.

Compile Kernel We first download source code for the kernel from its official website. Then we install dependency and development tools with a specific command. We extract the files to /home/seed/work/ directory. Then, we copy config from /boot in the kernel directory and type `sudo make -j$(nproc)` to build the kernel image and modules. Last, we only need to install the kernel modules and reboot to load it. However, later while coding the program 2, it requires us to use the unexported symbol from specific libraries. To export it, we simply go to /home/seed/work/KERNEL_FILE directory and find the specific c files that contain the function we want to use. We put `EXPORT_SYMBOL(FUNC_NAME);` after the function is declared. To make it usable, we need to recompile the kernel by starting from the kernel image and modules build.

Execution Steps

Program 1

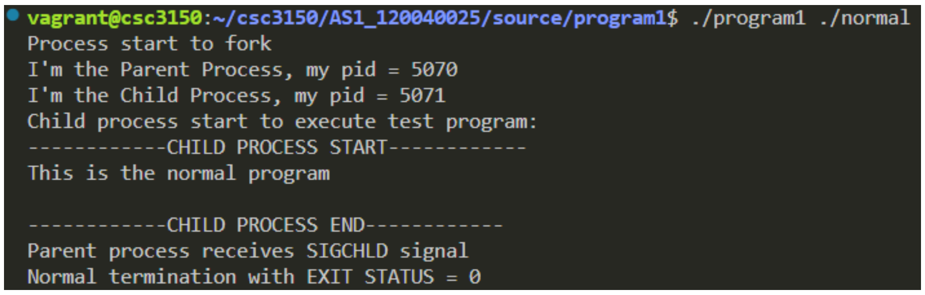
- Open a new terminal and type `cd {path}` to enter the directory of the program 1.
- Type `make` to compile all the programs, including the main program and the test programs.
- Type `./program1 {test program}` to execute the main program along with the test program.

Program 2

- Modify the path for the test program in the program2.c source code. The default path is set to `/tmp/test`.
- Open a new terminal and type `cd {path}` to enter the directory of the program 2.
- Type `make` to compile the main program.
- Type `gcc -o test test.c` to compile the test program.
- Type `sudo insmod program2.ko` to insert the program2.ko kernel module into the local system module. To remove the program2.ko module from the local system module, type `sudo rmmod program2.ko`.
- Type `dmesg` to display the results. To show the 10 latest results, type `dmesg | tail -10`

Result

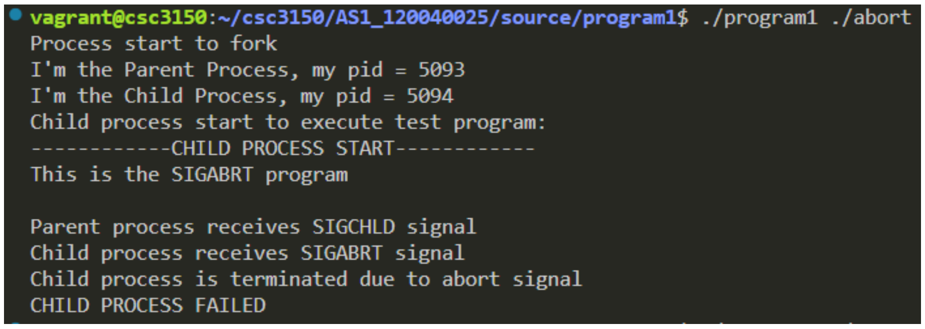
Program 1



```
vagrant@csc3150:~/csc3150/AS1_120040025/source/program1$ ./program1 ./normal
Process start to fork
I'm the Parent Process, my pid = 5070
I'm the Child Process, my pid = 5071
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
```

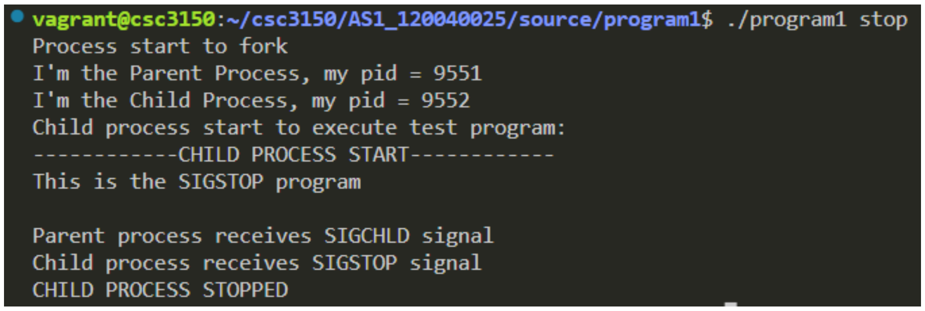
normal termination



```
vagrant@csc3150:~/csc3150/AS1_120040025/source/program1$ ./program1 ./abort
Process start to fork
I'm the Parent Process, my pid = 5093
I'm the Child Process, my pid = 5094
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGABRT program

Parent process receives SIGCHLD signal
Child process receives SIGABRT signal
Child process is terminated due to abort signal
CHILD PROCESS FAILED
```

signaled abort

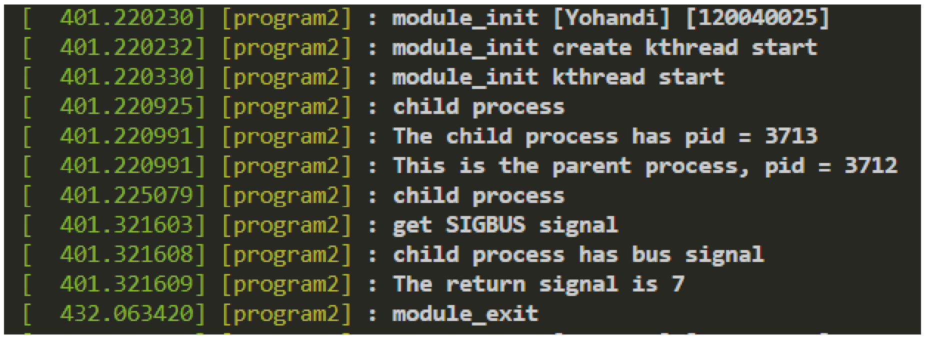


```
vagrant@csc3150:~/csc3150/AS1_120040025/source/program1$ ./program1 stop
Process start to fork
I'm the Parent Process, my pid = 9551
I'm the Child Process, my pid = 9552
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receives SIGCHLD signal
Child process receives SIGSTOP signal
CHILD PROCESS STOPPED
```

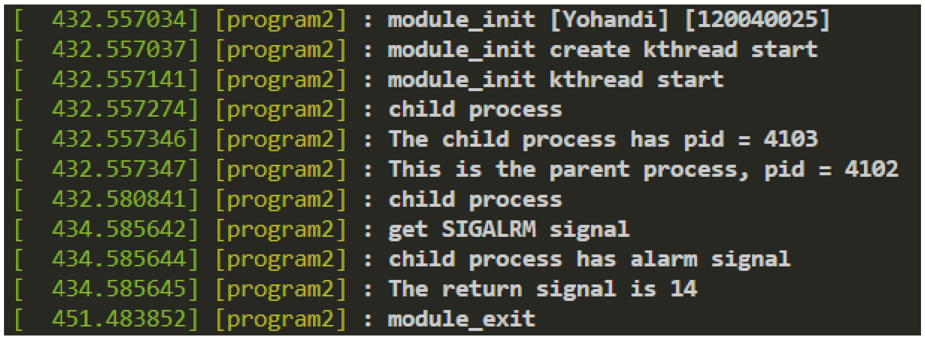
stopped

Program 2



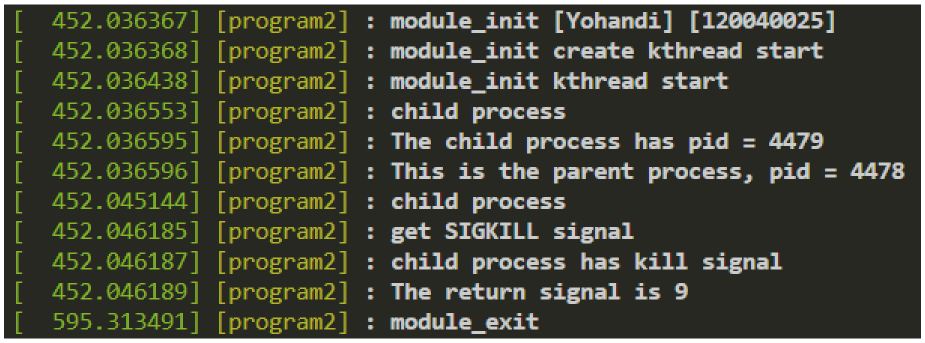
```
[  401.220230] [program2] : module_init [Yohandi] [120040025]
[  401.220232] [program2] : module_init create kthread start
[  401.220330] [program2] : module_init kthread start
[  401.220925] [program2] : child process
[  401.220991] [program2] : The child process has pid = 3713
[  401.220991] [program2] : This is the parent process, pid = 3712
[  401.225079] [program2] : child process
[  401.321603] [program2] : get SIGBUS signal
[  401.321608] [program2] : child process has bus signal
[  401.321609] [program2] : The return signal is 7
[  432.063420] [program2] : module_exit
```

signaled bus



```
[  432.557034] [program2] : module_init [Yohandi] [120040025]
[  432.557037] [program2] : module_init create kthread start
[  432.557141] [program2] : module_init kthread start
[  432.557274] [program2] : child process
[  432.557346] [program2] : The child process has pid = 4103
[  432.557347] [program2] : This is the parent process, pid = 4102
[  432.580841] [program2] : child process
[  434.585642] [program2] : get SIGALRM signal
[  434.585644] [program2] : child process has alarm signal
[  434.585645] [program2] : The return signal is 14
[  451.483852] [program2] : module_exit
```

signaled alarm



```
[  452.036367] [program2] : module_init [Yohandi] [120040025]
[  452.036368] [program2] : module_init create kthread start
[  452.036438] [program2] : module_init kthread start
[  452.036553] [program2] : child process
[  452.036595] [program2] : The child process has pid = 4479
[  452.036596] [program2] : This is the parent process, pid = 4478
[  452.045144] [program2] : child process
[  452.046185] [program2] : get SIGKILL signal
[  452.046187] [program2] : child process has kill signal
[  452.046189] [program2] : The return signal is 9
[  595.313491] [program2] : module_exit
```

signaled kill

Reflection

Program 1 taught us how to perform the forking procedure. It excites me that it is possible to simultaneously run two different processes. This means that we will execute two programs at the same time, which is seemingly not possible. Moreover, program 2 taught us a way to get through the wall between the kernel and users and attempt to code directly into the kernel. This makes us realize how a developer constructs their system. Although aside from these concerns, I am particularly interested in the bonus question, I haven't managed to solve it properly. It struggles me when trying to print out the Linux tree process. Despite not being able to complete the whole solution, it already inspires me to implement these lessons into another activity outside school.