CSC3150-OS-AS1-2024

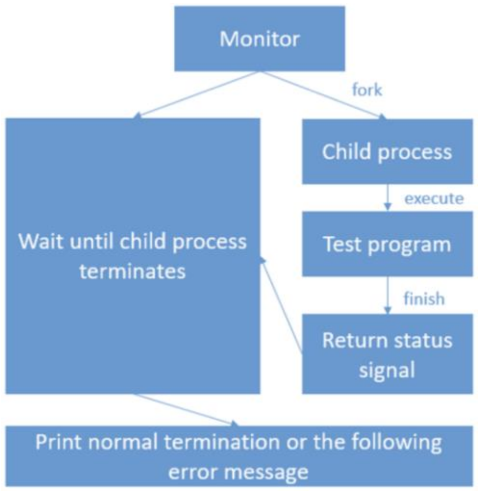
Kernel Mode Programming

Filbert Hamijoyo 122040012

Program Design

Program 1:

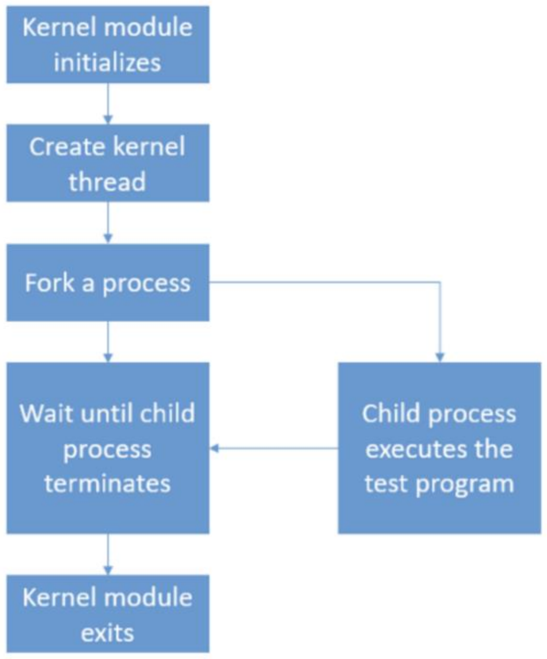
The design of Program 1 focuses on creating a child process from a parent process through the fork() system call. The fork() function returns an integer that determines the code path: a value of 0 means the child process is executing, and a positive value indicates the parent process is running. If the return value is negative, it means the forking has failed. In the child process, we utilize the execvp() function to replace the child process's memory space with a new process image, which is the specified test program. This function executes the test program passed as an argument. If execvp() fails, the error is captured, and the child process terminates with an error message. Additionally, before executing the test program, I used the raise(SIGSTOP) function to stop the child process and ensure the parent has a chance to handle the signals before the test program runs. The parent process, meanwhile, uses the waitpid() function to wait for the child's status to change. This is essential for monitoring whether the child process terminates normally or abnormally. The WIFEXITED() macro is used to check if the child exited normally, while WIFSIGNALED() and WIFSTOPPED() macros are employed to determine if the child was terminated by a signal or stopped. Depending on the outcome, the parent process retrieves the child's exit status or the signal that stopped/terminated the child and prints appropriate messages. By structuring the program this way, we simulate how processes and signals interact in a Unix-based operating system, handling real-world scenarios of process termination, signal sending, and resource cleanup. Flow chart provided for this task is the following:


Monitor
fork
Child process
execute
Test program
finish
Return status signal
Wait until child process terminates
Print normal termination or the following error message

Program 2:

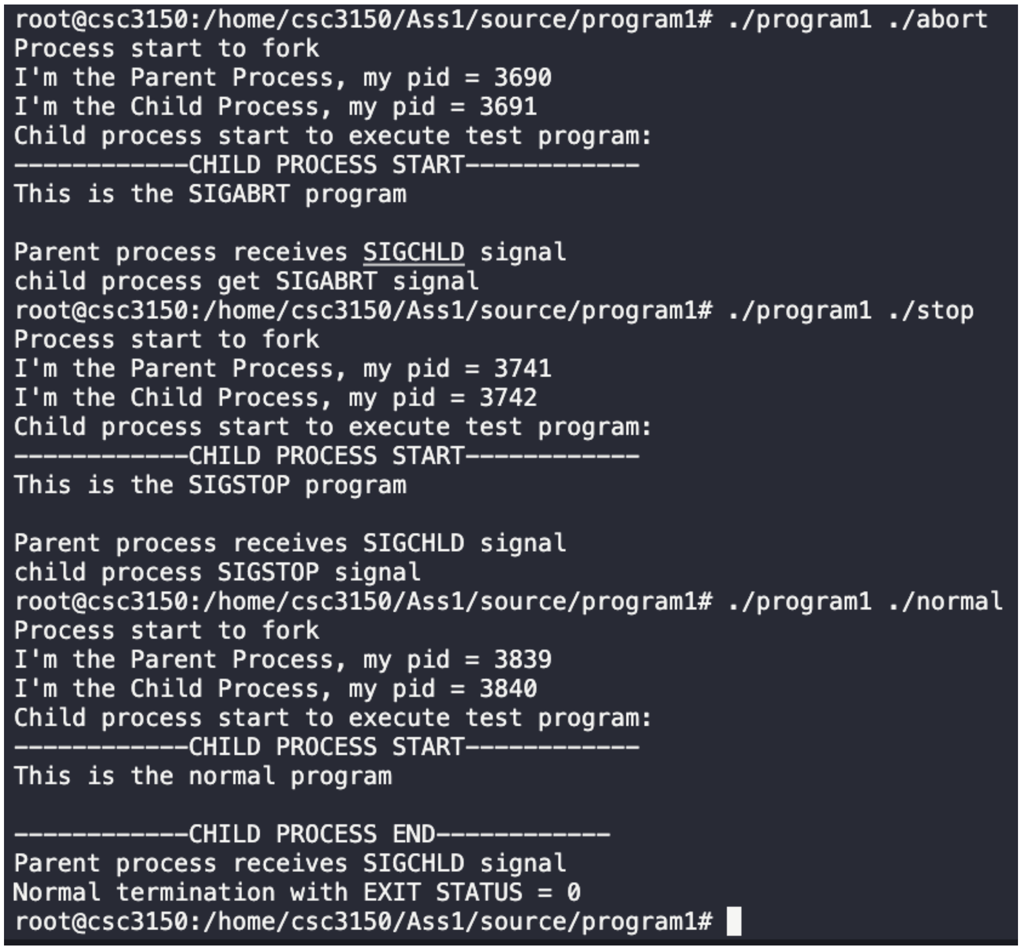
Program 2 expands on the concept of process management by diving into kernel-space operations. Here, we create a kernel module that handles process forking and signal management directly within the kernel. The core function, my_fork(), utilizes kernel_clone() to fork a child process, and the parent process waits for the child's termination using a custom waiting function my_wait(). Unlike Program 1, where the child process is created in user-space, Program 2's child process is created in kernel-space. In the child process, the my_exec() function executes an external program specified by a path using the do_execve() system call. This call directly replaces the current process image with the specified executable. Any failure in executing the program is handled with appropriate error messages. In the parent process, the do_wait() function is responsible for monitoring the child's termination status. Once the child process terminates or receives a signal, my_wait() evaluates the status and retrieves the signal number. I implemented a signal-mapping function get_signal_info() that converts the signal number to a human-readable name and description, providing clarity on how the child process was terminated (e.g., SIGINT, SIGSEGV, etc.). This allows the parent process to print detailed logs about the child's termination status and signal reception. By implementing these two programs, I explored both user-space and kernel-space process management, gaining a deeper understanding of how processes are handled at different layers of the operating system. Flow chart provided for this task is the following:

The main flow chart for Task 2 is:


Kernel module initializes
Create kernel thread
Fork a process
Wait until child process terminates
Child process executes the test program
Kernel module exits

Result

Program 1:



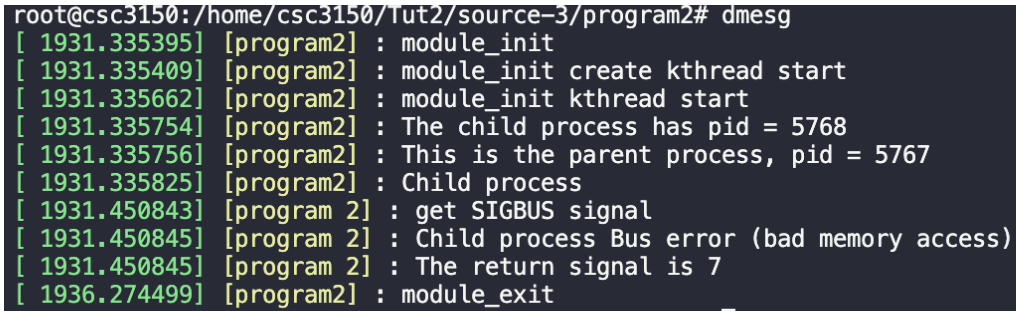
```
root@csc3150:/home/csc3150/Ass1/source/program1# ./program1 ./abort
Process start to fork
I'm the Parent Process, my pid = 3690
I'm the Child Process, my pid = 3691
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGABRT program

Parent process receives SIGCHLD signal
child process get SIGABRT signal
root@csc3150:/home/csc3150/Ass1/source/program1# ./program1 ./stop
Process start to fork
I'm the Parent Process, my pid = 3741
I'm the Child Process, my pid = 3742
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receives SIGCHLD signal
child process SIGSTOP signal
root@csc3150:/home/csc3150/Ass1/source/program1# ./program1 ./normal
Process start to fork
I'm the Parent Process, my pid = 3839
I'm the Child Process, my pid = 3840
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
root@csc3150:/home/csc3150/Ass1/source/program1#
```

Program 2:



```
root@csc3150:/home/csc3150/Tut2/source-3/program2# dmesg
[ 1931.335395] [program2] : module_init
[ 1931.335409] [program2] : module_init create kthread start
[ 1931.335662] [program2] : module_init kthread start
[ 1931.335754] [program2] : The child process has pid = 5768
[ 1931.335756] [program2] : This is the parent process, pid = 5767
[ 1931.335825] [program2] : Child process
[ 1931.450843] [program 2] : get SIGBUS signal
[ 1931.450845] [program 2] : Child process Bus error (bad memory access)
[ 1931.450845] [program 2] : The return signal is 7
[ 1936.274499] [program2] : module_exit
```

Development Environment Setup

System Setup:

To develop and run the programs, I worked within a Linux environment. Specifically, I used Ubuntu with the following versions:

- Linux Kernel Version: Linux 5.10.195
- GCC Version: GCC 5.4.0

Development Tools Installation:

To set up my development environment, I began by downloading UTM and the system archives from the official OS website. After successfully setting up UTM, I ran the CSC3150 virtual machine (VM) and used the SSH extension in Visual Studio Code (VSCode) to connect to the VM. Once inside the VM, I logged in as a superuser (sudo su) to gain the necessary privileges. I downloaded Linux Kernel 5.10.195 using wget, installed essential dependencies using apt-get, and unpacked the downloaded kernel using tar xvf. After organizing the kernel files in the appropriate directories, I configured the kernel using make mrproper, make clean, and make menuconfig. Once the kernel was configured, I installed the bc package and started the kernel compilation with make -j$(nproc), followed by installing the kernel modules using make modules_install and rebooting the system. To verify the kernel installation, I checked the version with uname -r. For Program 1, I compiled it with make and ran it using ./program1 $TEST_CASE. For Program 2, I compiled the module with make, inserted it using insmod program2.ko, and removed it with rmmod program2.ko. The output was checked through dmesg in the terminal, displaying the results from kernel-space.

Reflection

From these tasks, I learned a great deal about both user-space and kernel-space programming, specifically in relation to process creation and signal handling. **Program 1** provided a solid foundation in using the fork() and execvp() system calls, which are essential for creating and managing processes in Unix-like operating systems. I gained a deeper understanding of how the parent process monitors the child's termination or signal reception through the waitpid() system call and various macros like WIFEXITED() and WIFSIGNALED(). In **Program 2**, I delved into kernel-space programming, where processes are created and managed using system calls like kernel_clone() and do_execve(). I also implemented signal handling in the kernel by mapping signal numbers to their respective names and descriptions using custom functions. This allowed me to see how kernel-space process management differs from user-space and provided me with hands-on experience in managing processes and signals at a low system level. Furthermore, the opportunity to modify and recompile the Linux kernel to export unexported symbols expanded my understanding of kernel development, as I learned how to tailor the kernel environment to meet specific programming requirements. Overall, these tasks have significantly enhanced my knowledge of both user-space and kernel-space programming.