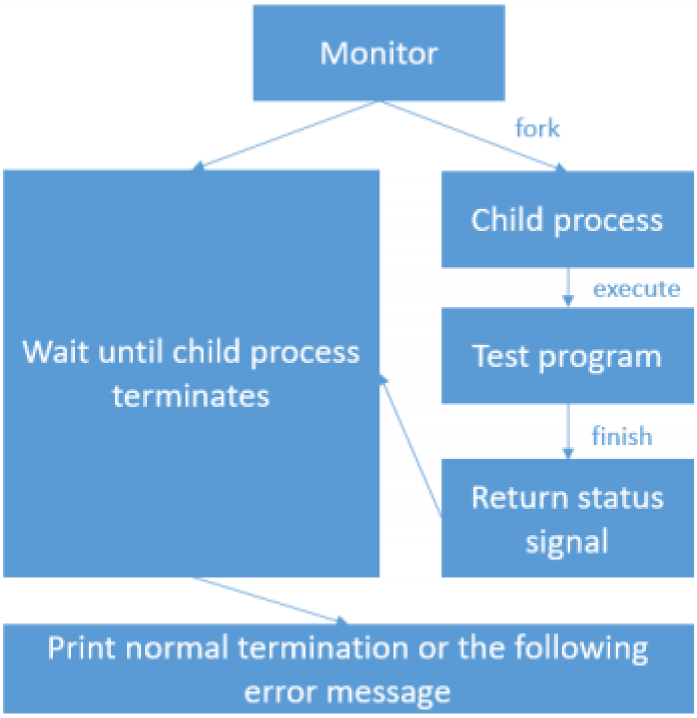
CSC3150 Assignment 1 Report

CHEN Ang (118010009)

Program Design

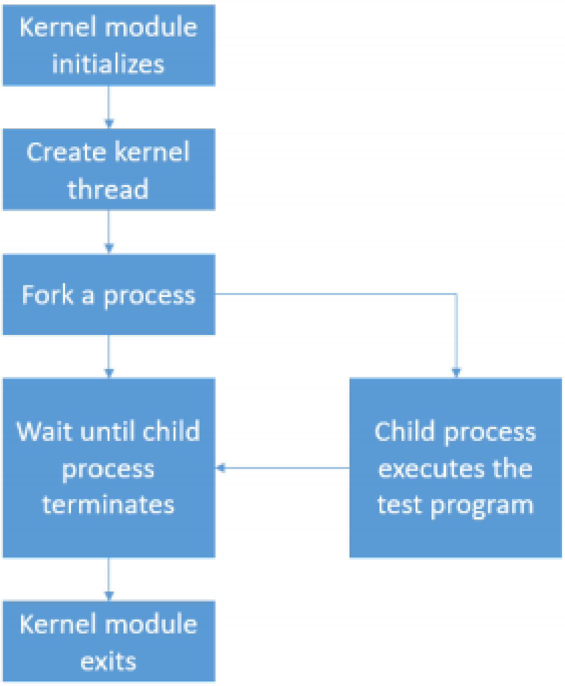
`program1.c` utilizes Linux system call APIs (`fork()`, `execve()`, `waitpid()`) to fork a child process which executes a user program, while the parent process wait for the child to terminate. The parent process then prints out the termination signal and status sent by its child.

- Flow chart of `program1.c` taken from the official project description:


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

`program2.c` implements a kernel module. The module creates a kernel thread and calls function `my_fork()`, a wrapper function for `_do_fork()` which forks a child process to execute `test` via `my_execve`, which calls `do_excve`. The parent process then does similar things as in `program_1.c`, except that it uses `my_wait(),` a wrapper of `do_wait` to wait for the child process, and that the termination signal and status is written into kernel log with `printk()`. Since the program is to be run as a kernel module, we are restricted to existing kernel modules, namely `_do_fork()`, `do_execve()`, `do_wait()`, and `getname()`.

- Flow chart of `program2.c` taken from the official project description:

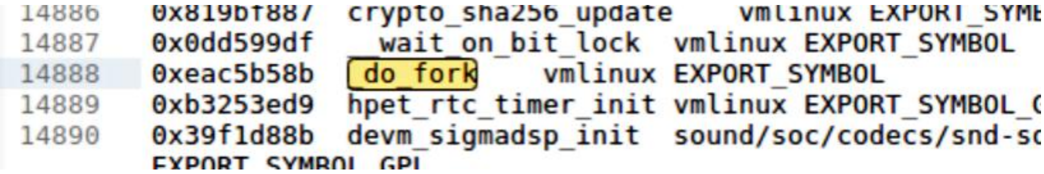

Kernel module initializes
Create kernel thread
Fork a process
Wait until child process terminates
Child process executes the test program
Kernel module exits

Environment

All programs are run on Ubuntu 16.04.2 LTS (32 bit), kernel version 4.10.14. Four symbols `_do_fork`, `do_execve`, `do_wait`, and `getname` are exported including `EXPORT_SYMBOL()` in respective source files before the kernel is recompiled and installed.

```
EXPORT_SYMBOL(_do_fork);
```

- You should see the symbols exported appear in `Module.symvers` after kernel has been rebuilt:



```
0x819bf887   crypto_sha256_update     vmlinux EXPORT_SYMB
0x0dd599df   __wait_on_bit_lock   vmlinux EXPORT_SYMBOL
0xeac5b58b   _do_fork      vmlinux EXPORT_SYMBOL
0xb3253ed9   hpet_rtc_timer_init vmlinux EXPORT_SYMBOL_G
0x39f1d88b   devm_sigmadsp_init   sound/soc/codecs/snd-sc
EXPORT_SYMBOL_GPL
```

How to Run the Programs

Both program folders contain a Makefile which allows quick compilation. Type `make` (or `make all`) in the terminal to compile all necessary files.

To execute `program1`, type in

```
./program1 [EXEC]
```

where `[EXEC]` is the name of the executable you wish to run by child process.

To execute `program2`, insert the kernel module by typing in

```
insmod program2.ko
```

You can then check the kernel log to see effects of the module

```
dmesg
```

or check last `[n]` logs

```
dmesg | tail -[n]
```

To remove the module, type in

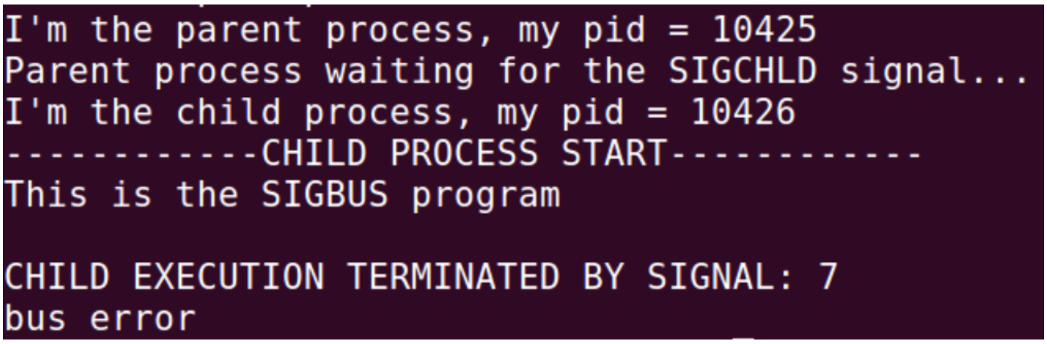
```
rmmod program2
```

You may `make clean` to clean all compiled files.

NOTICE: The variable `path` in `program2.c`-`my_exec()` should be modified to the absolute path of the test executable on your machine!

Screenshots

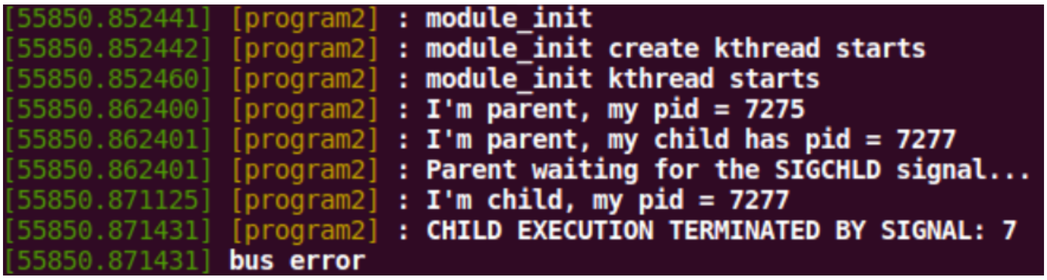
- Demo of program 1. The test program is `bus`



```
I'm the parent process, my pid = 10425
Parent process waiting for the SIGCHLD signal...
I'm the child process, my pid = 10426
------------CHILD PROCESS START------------
This is the SIGBUS program

CHILD EXECUTION TERMINATED BY SIGNAL: 7
bus error
```

- Demo of program 2. The program executed by child raises `SIGBUS`.



```
[55850.852441] [program2] : module_init
[55850.852442] [program2] : module_init create kthread starts
[55850.852460] [program2] : module_init kthread starts
[55850.862400] [program2] : I'm parent, my pid = 7275
[55850.862401] [program2] : I'm parent, my child has pid = 7277
[55850.862401] [program2] : Parent waiting for the SIGCHLD signal...
[55850.871125] [program2] : I'm child, my pid = 7277
[55850.871431] [program2] : CHILD EXECUTION TERMINATED BY SIGNAL: 7
[55850.871431] bus error
```

What I Learned

I learned the basics of processes, threads, forking, and how to write/compile/insert/remove a simple kernel module in Linux system. I also learned how to export symbols and how to compile the kernel. I learned how processes use signals to communicate and how the signals are decoded. I learned to see running programs from two perspectives: the perspective of the user in user land and the perspective of the OS in kernel land, and how they could change our ways of programming.

BONUS QUESTION

(Sorry for separating the bonus. I exported pdf too early and lost my md file :/)

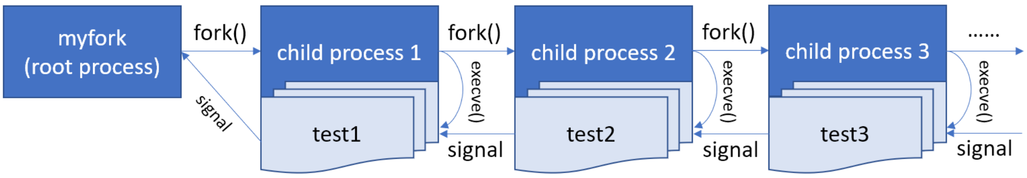
Program Design

In the bonus we implemented `myfork` which executes multiple executables as input arguments in a forking chain, that is, with the command

```
./myfork test1 test2 ...
```

`myfork` will fork a child process to execute `test1`, that child process thereafter forks another child process to execute `test2`, and so on.

- Flow chart of `myfork.c`, originally created:



We count the number of arguments `argc` to determine the number of `fork()` in a definite for loop. We only do `fork()` when the `pid` returned by the previous `fork()` (or that of the root process, which is initialized at 0) is 0, indicating the current child process. After each forking the child process becomes a parent and we let the process wait for its child just forked. When the loop finishes, we end of at the last child process which has no further forking to do. So it simply `execve()` the last executable. We then go through a backward pass which signals the parent processes one by one, each executing an executable, until the very root process, which we have no `execve()` to do.

Environment

The environment of this program is the same as the `program1`.

How to Run the Program

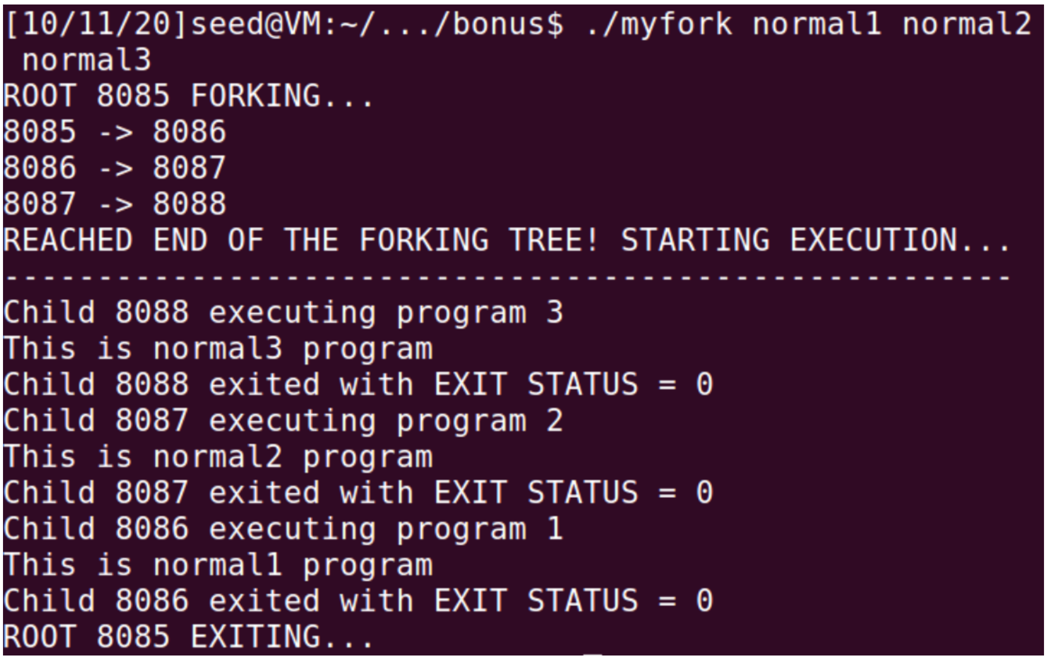
Run `make` to compile all files necessary.

To let `myfork` spawn a forking tree of, say, `myfork -> [TEST1] -> [TEST2]`, type in

```
./myfork [TEST1] [TEST2]
```

where `[TEST1], [TEST2]` are the testing executables to be run by the child processes.

Screenshot



```
[10/11/20]seed@VM:~/.../bonus$ ./myfork normal1 normal2
 normal3
ROOT 8085 FORKING...
8085 -> 8086
8086 -> 8087
8087 -> 8088
REACHED END OF THE FORKING TREE! STARTING EXECUTION...
------------------------------------------------------
Child 8088 executing program 3
This is normal3 program
Child 8088 exited with EXIT STATUS = 0
Child 8087 executing program 2
This is normal2 program
Child 8087 exited with EXIT STATUS = 0
Child 8086 executing program 1
This is normal1 program
Child 8086 exited with EXIT STATUS = 0
ROOT 8085 EXITING...
```

What I Learned

I learned how we chain `fork()` commands together and how to keep track of one particular child process at a time.