CSC3150 Assignment1 Report

Name: Xiang Fei

Student ID: 120090414

1. Design

1a. Overview

Assignment 1 asks us to do some programming exercise about process in user mode and kernel mode. In this assignment, we need to learn how to install the virtual machine and deploy the kernel environment. Besides, we are required to understand the working logic of process and kernel. And for bonus question, we need to learn the proc filesystem, how to get the pid, ppid, and other information of a process.

1b. Task 1

Task 1 is quite simple, since it is based on user mode. In this task, we need to fork a child process, and then, execute the test program and get the status signal information from child process. Finally, we are required to print out this information in parent process. The pivotal API that I used in this task is **fork()**, **execve()**, **waitpid()**, etc. First, I use **fork()** to generate a child process. Then, **execve()** is used in child process to execute the test program. During the execution, **waitpid()** is used in parent process to wait for child process terminates and get the returned signal information. Finally, in parent process, I use **WIFEXITED()**, **WEXITSTATUS()** and **WIFSTOPPED()** to check the received signal and print out the information.

1c. Task 2

Task 2 is more difficult than task 1, since it is based on kernel mode, which means that we need to deploy the correct kernel environment and implement the same function in a more underlying way. The detailed of deploying the environment will be discussed in later. Here, I just focus on the functions we need to implement in task 2. First, I write **myfork()** function to create a kernel thread. In fact, this function is based on two functions **my_exec()** and **my_wait()**, which are also written by myself. **my_exec()** function is used for child process to executing the test program, which is based on **do_execve()** and **getname_kernel** (get the filename of the test program). For the parent process, it will wait until the child process terminates using **my_wait()** function, which is based on **do_wait()** and combine the construction of the wait_opts structure (since it is not included in the head file, we need to write this structure by ourselves), and then print the signal information to kernel log. Similar to task 1, I classify the signal according to its value, here is the member variable wo_stat of wait_opts structure and we need to use (& 0x7f) since the value we get from wo_stat is the true signal shifted for some signals. Finally, exit the module. Since we need to use the functions in kernel source code, we are required to export the symbol in kernel source code and extern them in our script, including **kernel_clone**, **do_execve**, **getname_kernel** and **do_wait**. After the modification, recompilation of the kernel is needed. The updated modules need to be installed.

1d. Bonus Task

In this task, we need to write a code to implement the **pstree** function in linux commend and the options of this function. I will introduce my implementation in detail. First, I construct a structure call **TNode** to store the

information of each process and thread, including PID, PPID, TGID, process name, the child process pointer, the peer process pointer, the boolean type variables checkFirstChild (whether it is the first child in the next generation, which influences the print logic), InThreadGroup (whether it is in thread group, which also influences the print logic, whether we need to use {} when print out the process name), HasBeenConsidered (it is used when building the node, if it has bee considered, then we don't need to consider it again). Here, the boolean type is defined by myself. Then, I get the process information in /proc directory, here I enter the number directory and read the /status file to get the detail information, such as pid, ppid, tgid, process name. And for thread, I get their information through /task, and tgid is their parent's pid. These information are first stored in arrays. And then, I build the tree by recursively invoke **buildNode** function. After I build the tree, I recursively invoke **PrintNode** function to print this tree like **pstree** commend. In fact, since the time is really limited and I want to learn this knowledge, so my implementation method is based on a github repository. But I still do some personal work to modify that code, for example, the coding logic, the printing format of the tree, and the options of this commend. I just want to show the elegant implementation and my understanding of this task, which is based on recursion method. TAs are not necessarily give me the full grade of bonus task since it is not totally my work. I implement the options -p and -V.

1e. Some details

1. The initialization of wait_opts object is like the following.

```
struct wait_opts wo;
  struct pid *wo_pid = NULL;
  enum pid_type type;
  type = PIDTYPE_PID;
  wo_pid = find_get_pid(pid);

  wo.wo_type = type;
  wo.wo_pid = wo_pid;
  wo.wo_flags = WEXITED | WSTOPPED; // it is used to deal with the stop signal
  wo.wo_info = NULL;
  wo.wo_rusage = NULL;
```

2. The initialization of kernel_clone_args object is like the following.

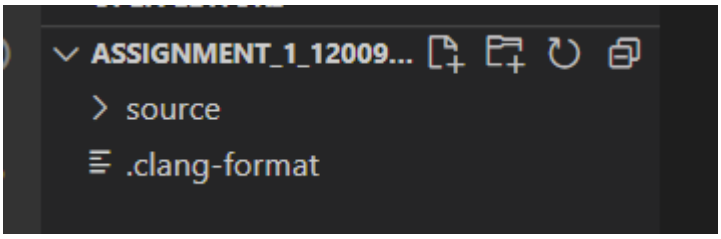
```
struct kernel_clone_args kargs = {
    .flags = ((SIGCHLD | CLONE_VM | CLONE_UNTRACED) & ~CSIGNAL),
    .pidfd = NULL,
    .child_tid = NULL,
    .parent_tid = NULL,
    .exit_signal = SIGCHLD & CSIGNAL,
    .stack = (unsigned long)&my_exec,
    .stack_size = 0,
    .tls = 0};
```

I know the values of the member variables by seeing the source code.

1f. Setting clang-format

The clang-format I used is the same as the kernel source code.

First, I copy the ".clang-format" file from the linux-kernel to my assignment directory.



And I install the clang-format-8 (the version is larger than or equal to 4, otherwise, there is some problem). Then, I use the commend like the following to change my code format to be the same as the kernel source code format.

```
vagrant@csc3150:~/Assignment_1_120090414$ cd source/program1
vagrant@csc3150:~/Assignment_1_120090414/source/program1$ clang-format-8 -i program1.c
vagrant@csc3150:~/Assignment_1_120090414/source/program1$ cd ..
vagrant@csc3150:~/Assignment_1_120090414/source$ cd program2
vagrant@csc3150:~/Assignment_1_120090414/source/program2$ clang-format-8 -i program2.c
vagrant@csc3150:~/Assignment_1_120090414/source/program2$ cd ..
vagrant@csc3150:~/Assignment_1_120090414/source$ cd bonus/
vagrant@csc3150:~/Assignment_1_120090414/source/bonus$ clang-format-8 -i pstree.c
vagrant@csc3150:~/Assignment_1_120090414/source/bonus$
```

2. Set up the environment

2a. Environment

OS version: Ubuntu 16.04.7 LTS

Kernel version: 5.10.27

gcc version: 5.4.0 (above 4.9, satisfy the requirement)

```
vagrant@csc3150:~/Assignment_1_120090414$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.04.7 LTS
Release:        16.04
Codename:       xenial
vagrant@csc3150:~/Assignment_1_120090414$ uname -r
5.10.27
```

```
Thread model: posix
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)
```

2b. Details of setting up the environment

The installation of virtual machine and the settings of vagrant, the compilation process is introduced very meticulously in tutorial, so I think I don't need to repeat the procedure here. The most important thing is my modification in kernel source code.

I need to export symbols of four functions "kernel_clone"(/kernel/fork.c), "do_execve" (/fs/exec.c), "getname_kernel" (/fs/namei.c) and "do_wait"(/kernel/exit.c). I use "sudo su" to get the permission to modify kernel code and use vim to modify it. The code is just like:

```
EXPORT_SYMBOL(do_execve);
```

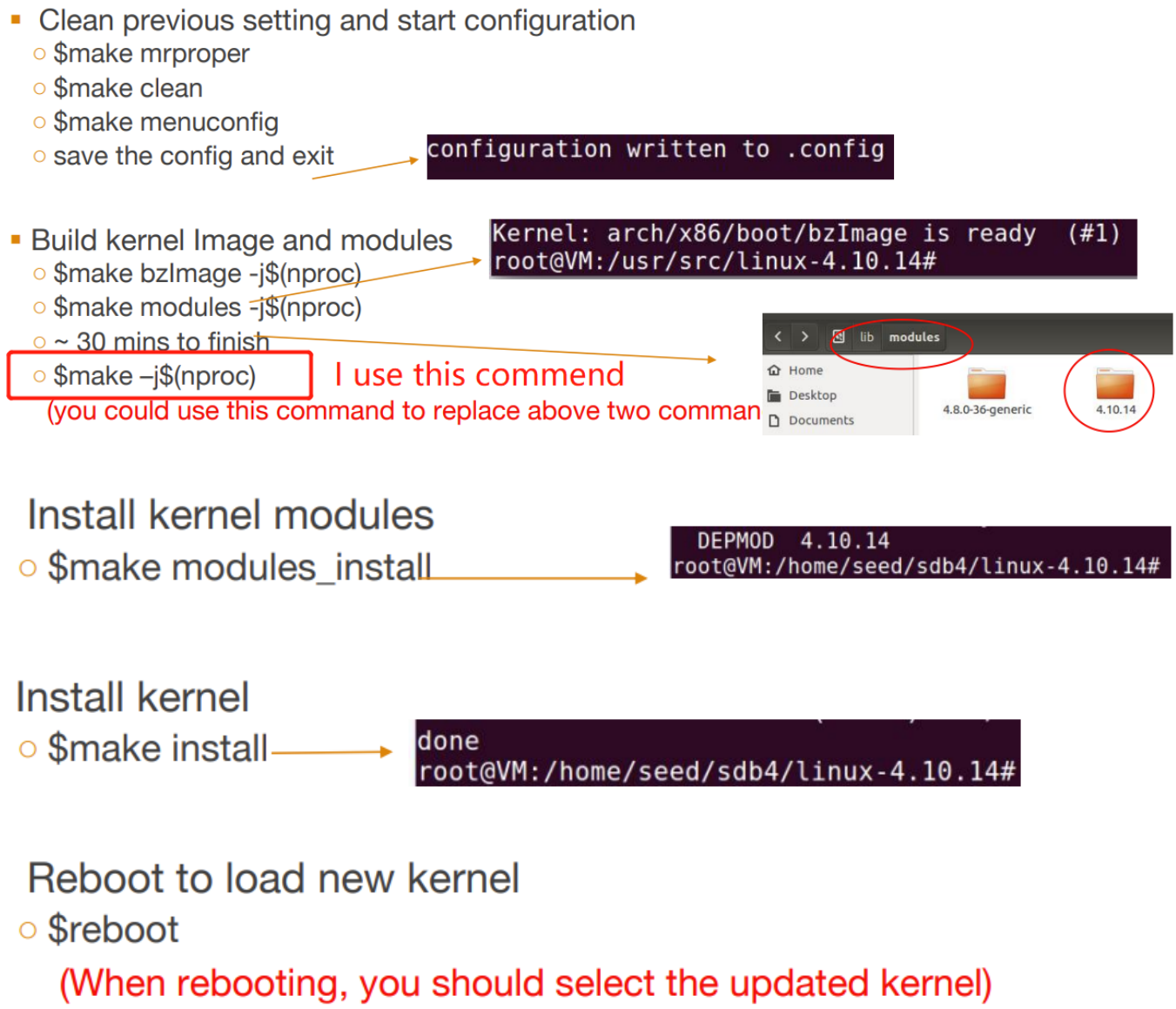
We should pay attention that the code should added after the inclusion of head files. And for static function, we need to delete the "static" keyword.

In our c program, we need to extern these functions, like the following:

```
extern pid_t kernel_clone(struct kernel_clone_args *kargs);
extern int do_execve(struct filename *filename,
                     const char __user *const __user *__argv,
                     const char __user *const __user *__envp);
extern long do_wait(struct wait_opts *wo);
extern struct filename *getname_kernel(const char *filename);
```

After modifying the kernel source code, we can compile the code. The procedure in tutorial is very detailed, like the following:

- Download source code from
  - http://www.kernel.org
  - mirro: https://mirror.tuna.tsinghua.edu.cn/kernel/v5.x/
- Install Dependency and development tools
  - sudo apt-get install libncurses-dev gawk flex bison openssl libssl-dev dkms libelf-dev libudev-dev libpci-dev libiberty-dev autoconf llvm dwarves
- Extract the source file to /home/seed/work
  - cp KERNEL_FILE.tar.xz /home/seed/work
  - cd /home/seed/work
  - $sudo tar xvf KERNEL_FILE.tar.xz
- Copy config from /boot to /home/seed/work/KERNEL_FILE
- Login root account and go to kernel source directory
  - $sudo su
  - $cd /home/seed/work /KERNEL_FILE



3. Execution

3a. Task 1

1. Cd to the directory of program 1 (containing all the source codes and makefile)
2. Type "make"
3. Type "./program1 testfile". testfile is the file to be executed in child process. For example, "./program1 alarm"

3b. Task 2

1. Update the kernel source code (add EXPORT_SYMBOL() for 4 functions invoked)
2. Compile the kernel: sudo su; cd to kernel file; make mrproper; make clean; make menuconfig (need to install a tool here); (Recompile start from here) make bzImage; make modules; make modules_install; make install;
3. reboot
4. Cd to the directory of program 2 (containing all the source codes and makefile).
5. Compile the test program: gcc –o test test.c
6. Type "sudo make"
7. Type "sudo insmod program2.ko" to install the module.

8. Type "sudo rmmod program2" to remove the module.
9. Type "dmesg" to get the kernel log.

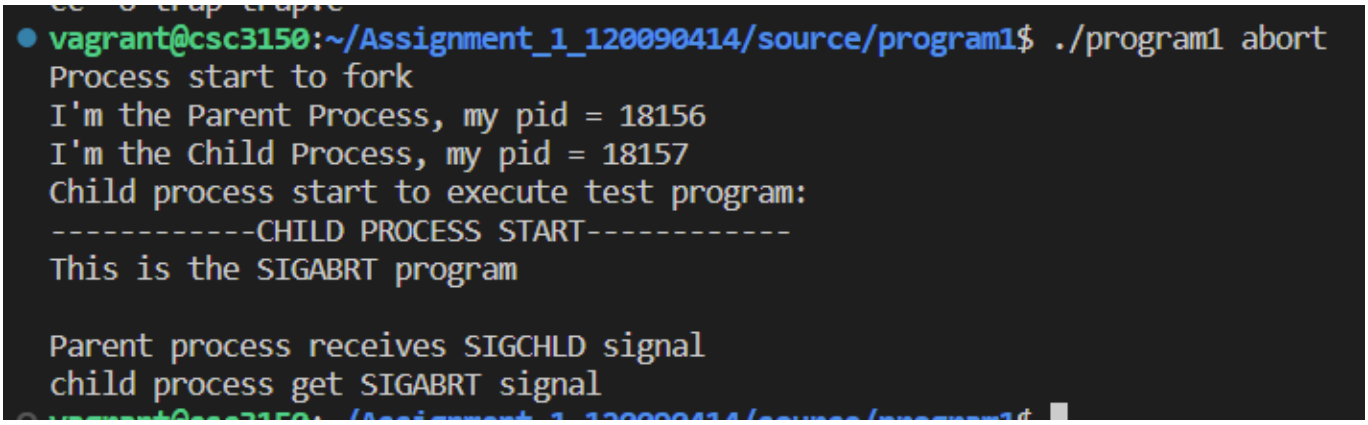
3c. Task 3

1. Cd to the directory of bonus (containing all the source codes and makefile)
2. Type "make"
3. Type "./pstree" to see the normal print result. Type "./pstree -V", "./pstree -p" to see the print result under options.

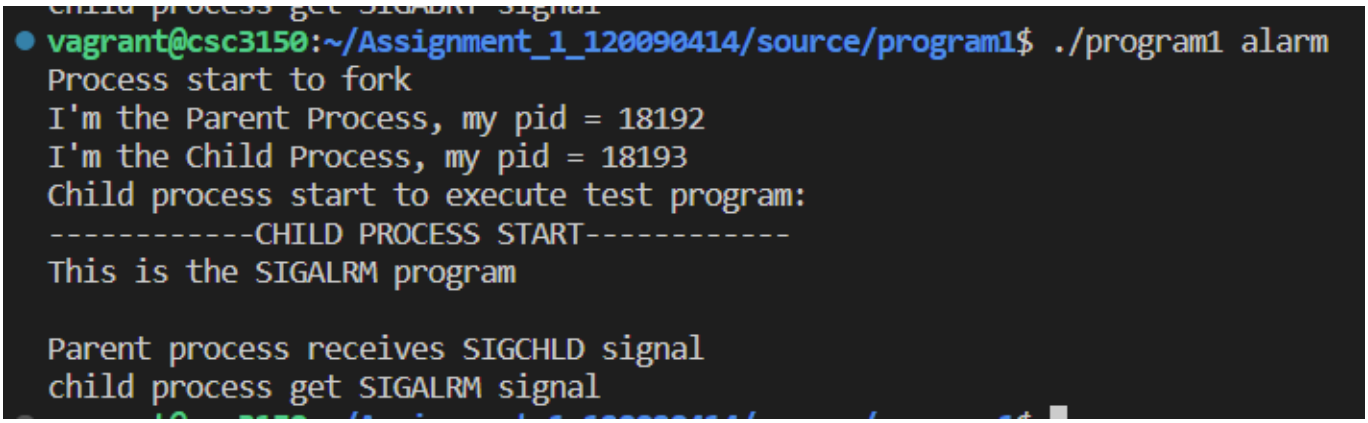
4. Output

4a. Task 1

1. Abort signal



2. Alarm signal



3. Bus signal

```
vagrant@csc3150:~/Assignment_1_120090414/source/program1$ ./program1 bus
Process start to fork
I'm the Parent Process, my pid = 18239
I'm the Child Process, my pid = 18240
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGBUS program

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

4. Floating signal

```
vagrant@csc3150:~/Assignment_1_120090414/source/program1$ ./program1 floating
Process start to fork
I'm the Parent Process, my pid = 18284
I'm the Child Process, my pid = 18285
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGFPE program

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

5. Hangup signal

```
vagrant@csc3150:~/Assignment_1_120090414/source/program1$ ./program1 hangup
Process start to fork
I'm the Parent Process, my pid = 18326
I'm the Child Process, my pid = 18327
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGHUP program

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

6. Illegal_instr signal

```
vagrant@csc3150:~/Assignment_1_120090414/source/program1$ ./program1 illegal_instr
Process start to fork
I'm the Parent Process, my pid = 18351
I'm the Child Process, my pid = 18352
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGILL program

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

7. Interrupt signal

```
vagrant@csc3150:~/Assignment_1_120090414/source/program1$ ./program1 interrupt
Process start to fork
I'm the Parent Process, my pid = 18398
I'm the Child Process, my pid = 18399
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGINT program

Parent process receives SIGCHLD signal
child process get SIGINT signal
vagrant@csc3150:~/Assignment_1_120090414/source/program1$
```

8. Kill signal

```
vagrant@csc3150:~/Assignment_1_120090414/source/program1$ ./program1 kill
Process start to fork
I'm the Parent Process, my pid = 18453
I'm the Child Process, my pid = 18454
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGKILL program

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

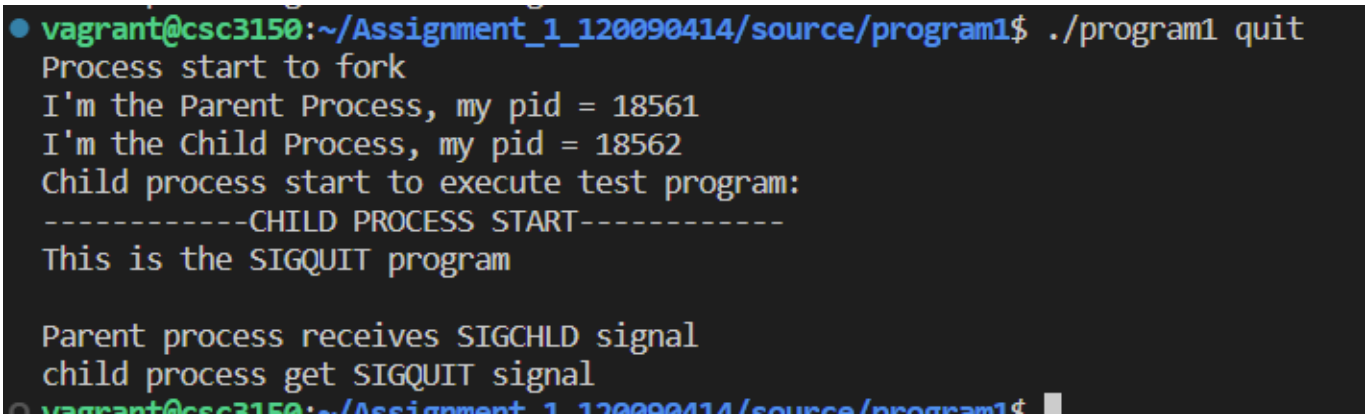
9. Normal signal

```
vagrant@csc3150:~/Assignment_1_120090414/source/program1$ ./program1 normal
Process start to fork
I'm the Child Process, my pid = 18500
Child process start to execute test program:
I'm the Parent Process, my pid = 18499
------------CHILD PROCESS START------------
This is the normal program

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

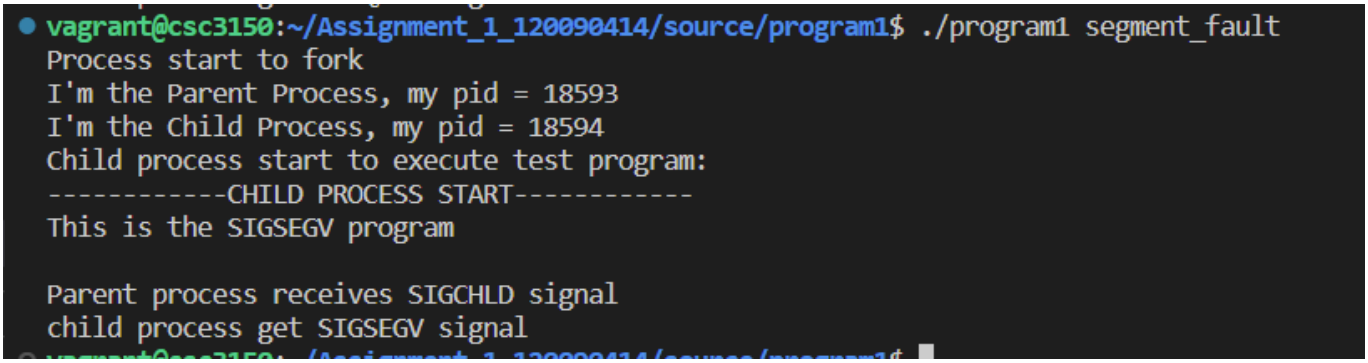
10. Pipe signal

```
vagrant@csc3150:~/Assignment_1_120090414/source/program1$ ./program1 pipe
Process start to fork
I'm the Parent Process, my pid = 18530
I'm the Child Process, my pid = 18531
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGPIPE program

Parent process receives SIGCHLD signal
child process get SIGPIPE signal
vagrant@csc3150:~/Assignment_1_120090414/source/program1$
```

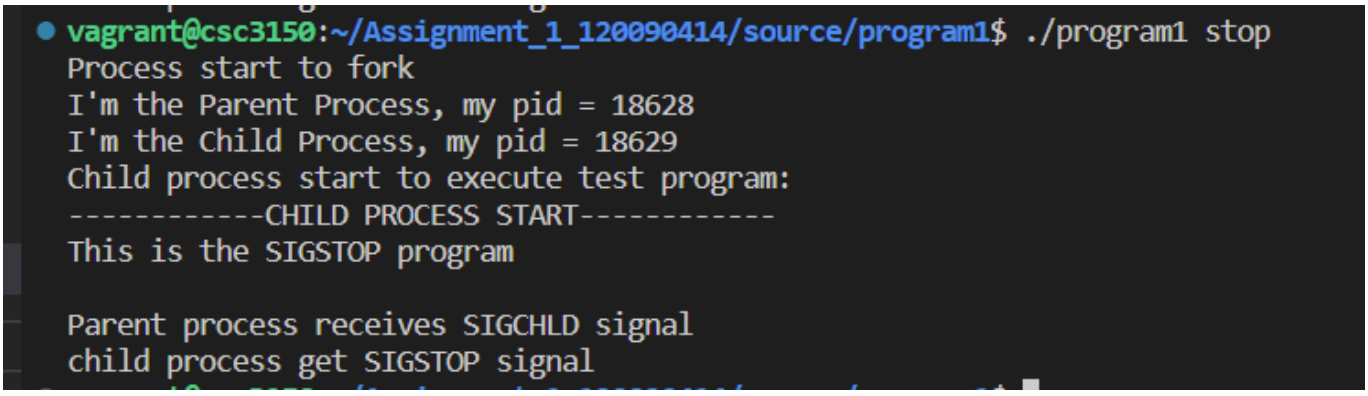
11. Quit signal



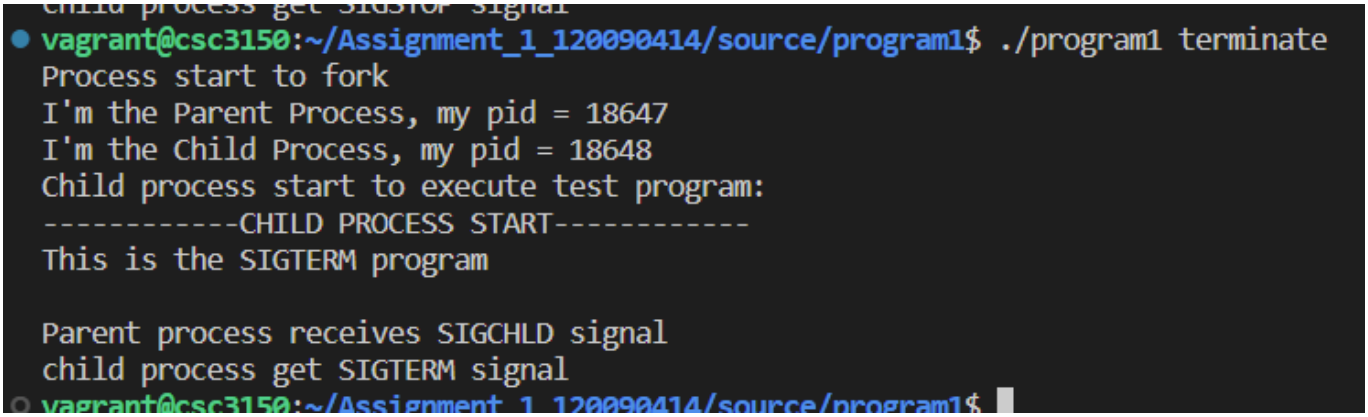
12. Segment_fault signal



13. Stop signal



14. Terminate signal



15. trap signal

```
vagrant@csc3150:~/Assignment_1_120090414/source/program1$ ./program1 trap
Process start to fork
I'm the Parent Process, my pid = 18679
I'm the Child Process, my pid = 18680
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGTRAP program

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

4b. Task 2

1. Abort signal

```
vagrant@csc3150:~/Assignment_1_120090414/source/program2$ dmesg
[42881.499340] [program2] : module_init Xiang Fei 120090414
[42881.499341] [program2] : module_init create kthread start
[42881.499436] [program2] : module_init kthread start
[42881.500415] [program2] : The child process has pid = 7171
[42881.500416] [program2] : This is the parent process, pid = 7169
[42881.500417] [program2] : child process
[42881.582033] [program2] : get SIGABRT signal
[42881.582034] [program2] : child process is aborted
[42881.582035] [program2] : The return signal is 6
[42884.487379] [program2] : module_exit./my
```

2. Alarm signal

```
[42934.998150] [program2] : module_init Xiang Fei 120090414
[42934.998151] [program2] : module_init create kthread start
[42934.998210] [program2] : module_init kthread start
[42934.999508] [program2] : The child process has pid = 7805
[42934.999509] [program2] : This is the parent process, pid = 7802
[42934.999750] [program2] : child process
[42937.035825] [program2] : get SIGALRM signal
[42937.035830] [program2] : child process is alarmed
[42937.035834] [program2] : The return signal is 14
[42937.473690] [program2] : module_exit./my
```

3. Bus signal

```
[42977.221173] [program2] : module_init Xiang Fei 120090414
[42977.221175] [program2] : module_init create kthread start
[42977.221246] [program2] : module_init kthread start
[42977.222033] [program2] : The child process has pid = 8445
[42977.222033] [program2] : This is the parent process, pid = 8443
[42977.222034] [program2] : child process
[42977.299985] [program2] : get SIGBUS signal
[42977.299986] [program2] : child process has bus error
[42977.299987] [program2] : The return signal is 7
[42979.656820] [program2] : module_exit./my
```

4. Floating signal

```
[43014.763396] [program2] : module_init kthread start
[43014.763519] [program2] : The child process has pid = 9086
[43014.763519] [program2] : This is the parent process, pid = 9085
[43014.763521] [program2] : child process
[43014.842537] [program2] : get SIGFPE signal
[43014.842538] [program2] : child process has floating point exception
[43014.842539] [program2] : The return signal is 8
[43017.129191] [program2] : module_exit./my
```

5. Hangup signal

```
[43052.337040] [program2] : module_init Xiang Fei 120090414
[43052.337042] [program2] : module_init create kthread start
[43052.337086] [program2] : module_init kthread start
[43052.337175] [program2] : The child process has pid = 9730
[43052.337175] [program2] : This is the parent process, pid = 9729
[43052.337177] [program2] : child process
[43052.337453] [program2] : get SIGHUP signal
[43052.337453] [program2] : child process terminal hangup or process death
[43052.337454] [program2] : The return signal is 1
[43055.341751] [program2] : module_exit./my
```

6. Illegal_instr signal

```
[45114.359616] [program2] : module_init Xiang Fei 120090414
[45114.359617] [program2] : module_init create kthread start
[45114.359664] [program2] : module_init kthread start
[45114.360957] [program2] : The child process has pid = 10484
[45114.360958] [program2] : This is the parent process, pid = 10482
[45114.360959] [program2] : child process
[45114.438842] [program2] : get SIGILL signal
[45114.438843] [program2] : child process has illegal instruction
[45114.438844] [program2] : The return signal is 4
[45116.817816] [program2] : module_exit./my
```

7. Interrupt signal

```
[45152.483446] [program2] : module_init Xiang Fei 120090414
[45152.483447] [program2] : module_init create kthread start
[45152.483466] [program2] : module_init kthread start
[45152.489242] [program2] : The child process has pid = 11116
[45152.489243] [program2] : This is the parent process, pid = 11113
[45152.490472] [program2] : child process
[45152.490789] [program2] : get SIGINT signal
[45152.490790] [program2] : child process receives interrupt signal from keyboard
[45152.490790] [program2] : The return signal is 2
[45154.703886] [program2] : module_exit./my
```

8. Kill signal

```
[45265.430816] [program2] : module_init Xiang Fei 120090414
[45265.430817] [program2] : module_init create kthread start
[45265.430906] [program2] : module_init kthread start
[45265.431441] [program2] : The child process has pid = 11774
[45265.431441] [program2] : This is the parent process, pid = 11772
[45265.431443] [program2] : child process
[45265.431650] [program2] : get SIGKILL signal
[45265.431650] [program2] : child process is killed
[45265.431651] [program2] : The return signal is 9
[45269.195708] [program2] : module_exit./my
```

9. Normal signal

```
[45357.868578] [program2] : module_init Xiang Fei 120090414
[45357.868579] [program2] : module_init create kthread start
[45357.868614] [program2] : module_init kthread start
[45357.868751] [program2] : The child process has pid = 12426
[45357.868751] [program2] : This is the parent process, pid = 12425
[45357.868752] [program2] : child process
[45357.868991] [program2] : Normal termination
[45357.868992] [program2] : The return signal is 0
[45362.175123] [program2] : module_exit./my
```

10. Pipe signal

```
[45492.580878] [program2] : module_init Xiang Fei 120090414
[45492.580879] [program2] : module_init create kthread start
[45492.580914] [program2] : module_init kthread start
[45492.580936] [program2] : The child process has pid = 13061
[45492.580937] [program2] : This is the parent process, pid = 13060
[45492.580939] [program2] : child process
[45492.581452] [program2] : get SIGPIPE signal
[45492.581453] [program2] : child process writes on broken pipe
[45492.581453] [program2] : The return signal is 13
[45495.685653] [program2] : module_exit./my
```

11. Quit signal

```
[45540.405596] [program2] : module_init Xiang Fei 120090414
[45540.405597] [program2] : module_init create kthread start
[45540.405638] [program2] : module_init kthread start
[45540.406678] [program2] : The child process has pid = 13680
[45540.406679] [program2] : This is the parent process, pid = 13678
[45540.406680] [program2] : child process
[45540.483833] [program2] : get SIGQUIT signal
[45540.483834] [program2] : child process has quit signal
[45540.483835] [program2] : The return signal is 3
[45543.731975] [program2] : module_exit./my
```

12. Segment_fault signal

```
[45584.279302] [program2] : module_init Xiang Fei 120090414
[45584.279303] [program2] : module_init create kthread start
[45584.279341] [program2] : module_init kthread start
[45584.279362] [program2] : The child process has pid = 14300
[45584.279363] [program2] : This is the parent process, pid = 14299
[45584.279365] [program2] : child process
[45584.359206] [program2] : get SIGSEGV signal
[45584.359207] [program2] : child process refers to invalid memory
[45584.359209] [program2] : The return signal is 11
[45586.910511] [program2] : module_exit./my
```

13. Stop signal

```
[45617.567431] [program2] : module_init Xiang Fei 120090414
[45617.567432] [program2] : module_init create kthread start
[45617.567496] [program2] : module_init kthread start
[45617.567517] [program2] : The child process has pid = 14932
[45617.567518] [program2] : This is the parent process, pid = 14931
[45617.567519] [program2] : child process
[45617.567708] [program2] : get SIGSTOP signal
[45617.567708] [program2] : child process is stopped
[45617.567709] [program2] : The return signal is 19
[45622.985211] [program2] : module_exit./my
```

14. Terminate signal

```
[45653.892688] [program2] : module_init Xiang Fei 120090414
[45653.892689] [program2] : module_init create kthread start
[45653.892737] [program2] : module_init kthread start
[45653.892778] [program2] : The child process has pid = 15584
[45653.892779] [program2] : This is the parent process, pid = 15583
[45653.892781] [program2] : child process
[45653.893102] [program2] : get SIGTERM signal
[45653.893102] [program2] : child process terminates
[45653.893103] [program2] : The return signal is 15
[45657.493098] [program2] : module_exit./my
vagrant@csc3150:~/Assignment_1_120090414/source/program2$
```

15. trap signal

```
[45684.874579] [program2] : module_init Xiang Fei 120090414
[45684.874581] [program2] : module_init create kthread start
[45684.874654] [program2] : module_init kthread start
[45684.875183] [program2] : The child process has pid = 16202
[45684.875184] [program2] : This is the parent process, pid = 16200
[45684.875186] [program2] : child process
[45684.952713] [program2] : get SIGTRAP signal
[45684.952715] [program2] : child process reaches a breakpoint
[45684.952715] [program2] : The return signal is 5
[45687.378872] [program2] : module_exit./my
vagrant@csc3150:~/Assignment_1_120090414/source/program2$
```

4c. Bonus Task

1. ./pstree

```
vagrant@csc3150:~/Assignment_1_120090414/source/bonus$ ./pstree
systemd─┬─lxcfs─┬─{lxcfs}
        │       ├─{lxcfs}
        │       ├─{lxcfs}
        │       └─{lxcfs}
        ├─rsyslogd─┬─{rs:main Q:Reg}
        │          ├─{in:imklog}
        │          └─{in:imuxsock}
        ├─accounts-daemon─┬─{gdbus}
        │                 └─{gmain}
        ├─atd
        ├─systemd-logind
        ├─cron
        ├─acpid
        ├─iscsid
        ├─iscsid
        ├─dbus-daemon
        ├─dhclient
        ├─systemd-udevd
        ├─lvmetad
        ├─systemd-journal
        ├─stop
        ├─VBoxService─┬─{automount}
        │             ├─{vmstats}
        │             ├─{memballoon}
        │             ├─{cpuhotplug}
        │             ├─{vminfo}
        │             ├─{timesync}
        │             ├─{control}
        │             └─{RTThrdPP}
        ├─systemd──(sd-pam)
        ├─polkitd─┬─{gdbus}
```

2. ./pstree -V

```
└─unattended-upgr───{gmain}
vagrant@csc3150:~/Assignment_1_120090414/source/bonus$ ./pstree -V
pstree (PSmisc) 22.21
Copyright (C) 1993-2009 Werner Almesberger and Craig Small

PSmisc comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it under
the terms of the GNU General Public License.
For more information about these matters, see the files named COPYING.
vagrant@csc3150:~/Assignment_1_120090414/source/bonus$
```

3. ./pstree -p

```
vagrant@csc3150:~/Assignment_1_120090414/source/bonus$ ./pstree -p
systemd(1)─┬─lxcfs(994)─┬─{lxcfs}(3561)
           │            ├─{lxcfs}(3560)
           │            ├─{lxcfs}(1003)
           │            └─{lxcfs}(1002)
           ├─rsyslogd(993)─┬─{rs:main Q:Reg}(1015)
           │               ├─{in:imklog}(1014)
           │               └─{in:imuxsock}(1013)
           ├─accounts-daemon(987)─┬─{gdbus}(1058)
           │                      └─{gmain}(1021)
           ├─atd(986)
           ├─systemd-logind(984)
           ├─cron(981)
           ├─acpid(977)
           ├─iscsid(975)
           ├─iscsid(974)
           ├─dbus-daemon(970)
           ├─dhclient(822)
           ├─systemd-udevd(404)
           ├─lvmetad(402)
           ├─systemd-journal(362)
           ├─stop(14932)
           ├─VBoxService(1348)─┬─{automount}(1360)
           │                   ├─{vmstats}(1359)
           │                   ├─{memballoon}(1355)
           │                   ├─{cpuhotplug}(1354)
           │                   ├─{vminfo}(1353)
           │                   ├─{timesync}(1352)
           │                   ├─{control}(1351)
           │                   └─{RTThrdPP}(1350)
           └─systemd(1150)───(sd-pam)(1151)
```

5. My feeling

I have gained a lot from this assignment. The process of completing this assignment was not smooth sailing, and many difficulties were encountered during the process. For example, the virtual machine and physical machine do not have enough memory at the beginning, which leads to problems with kernel compilation; the kernel version used in the demonstration in the tutorial is different from the version we need to use, and there is nothing about the usage of those interfaces on the Internet. Detailed introduction, which led me to read the source code myself to learn how to use those interfaces and the logic in them, which is quite difficult.

In this assignment, I learned how to install the virtual machine and deploy the kernel environment. Besides, I also understand the working logic of process and kernel. And for bonus question, I learned the proc filesystem, how to get the pid, ppid, and other information of a process. For me, since I have an internship about backend-development, I'm familiar with linux commend, but this is the first time for me to implement a linux commend by myself using c language. And before, I have never learned the knowledge about kernel. I find that I already touch some more basic knowledge about computer and programming. It helps me a lot.

Last but not least, I can feel that TAs are hard-working and laborious for this course. They have large workloads and will be disturbed by mountains of questions raised by students. Good luck to you!