CSC3150 Operating System

Assignment Report #1

Name: Lingpeng Chen

Student ID: 120090049

Date: 2022/10/10

The Chinese University of Hong Kong, Shenzhen

1. Program 1

a) Program design

As is shown in figure 1, the steps of the program can be mainly divided into three parts.

i. Fork a child process to execute the test program in user mode.

ii. When the child process finishes execution, the parent process will receive the SIGCHLD signal from the child process.

iii. After getting the signal, the parent process will print out the termination information or the following error message of the child process.

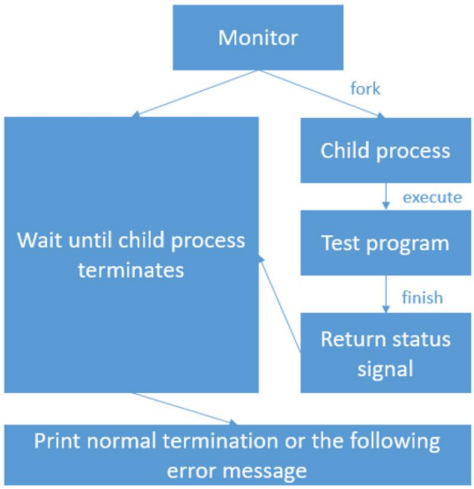


Fig 1. The main flow chart for Task 1

b) Detailed implementation

i. Fork a child process.

```
/* fork a child process */
int state;
printf("Process start to fork\n");
pid_t pid = fork();
```

ii. Execute the test program in the child process.

```
printf("I'm the Child Process, my pid = %d\n",
       getpid());
arg[argc - 1] = NULL;
for (i = 0; i < argc - 1; i++) {
    arg[i] = argv[i + 1];
}
printf("Child process start to execute test program:\n");
execve(arg[0], arg, NULL);
```

iii. Use waitpid() with the argument WUNTRACED so that the stop signal of the child process can be reported. Otherwise, the parent process will keep waiting for the child process.

```
waitpid(-1, &state, WUNTRACED);
printf("Parent process receives SIGCHLD signal\n");
```

iv. Use WITEXTED WIFSIGNALED WIFSTOPPED to determine the terminating state of the child process.

```
// Normal exit
if (WIFEXITED(state)) {···
// terminating signal
else if (WIFSIGNALED(state)) {···
// stop signal
else if (WIFSTOPPED(state)) {
    printf("child process get SIGSTOP signal\n");
} else {
    printf("continue\n");
}
exit(0);
```

v. Use the switch case statement to determine the type of the terminating signal.

```
else if (WIFSIGNALED(state)) {
    int signal_num = WTERMSIG(state);
    switch (signal_num) {
    // ./program1 ./abort #6
    case 6:
        printf("child process get SIGABRT signal\n");
        break;
    // ./program1 ./alarm #14
    case 14:
        printf("child process get SIGALRM signal\n");
        // printf("This is the SIGALRM signal\n");
        break;
    // ./program1 ./bus   #7
    case 7:
        printf("child process get SIGBUS signal\n");
        // printf("This is the SIGBUS signal\n");
        break;
```

c) Environment

In this task, the code is designed in the user mode. There is no need to change the environment, the only thing I need to do is to check the gcc and kernel version.

```
vagrant@csc3150:~/csc3150/Assignment_1_120090049$ cat /proc/version
Linux version 5.10.5 (root@csc3150) (gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609, GNU ld (GNU Binutils for Ubuntu) 2.26.1) #2 SMP Tue Oct 4 15:48:51 UTC 2022
```

d) Result

i. Output for normal termination

```
vagrant@csc3150:~/csc3150/Assignment_1_120090049/source/program1$ ./program1 ./normal
Process start to fork
I'm the Parant Process, my pid = 24588
I'm the Child Process, my pid = 24589
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
```

ii. Output for stopped

```
vagrant@csc3150:~/csc3150/Assignment_1_120090049/source/program1$ ./program1 ./stop
Process start to fork
I'm the Parant Process, my pid = 24691
I'm the Child Process, my pid = 24692
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
```

iii. Output for signaled abort/alarm/bus/floating/hangup/illegal_instr/interrupt/kill/pipe/quit/segement_fault/terminate/trap.
The result is shown in figure 2 and figure 3.

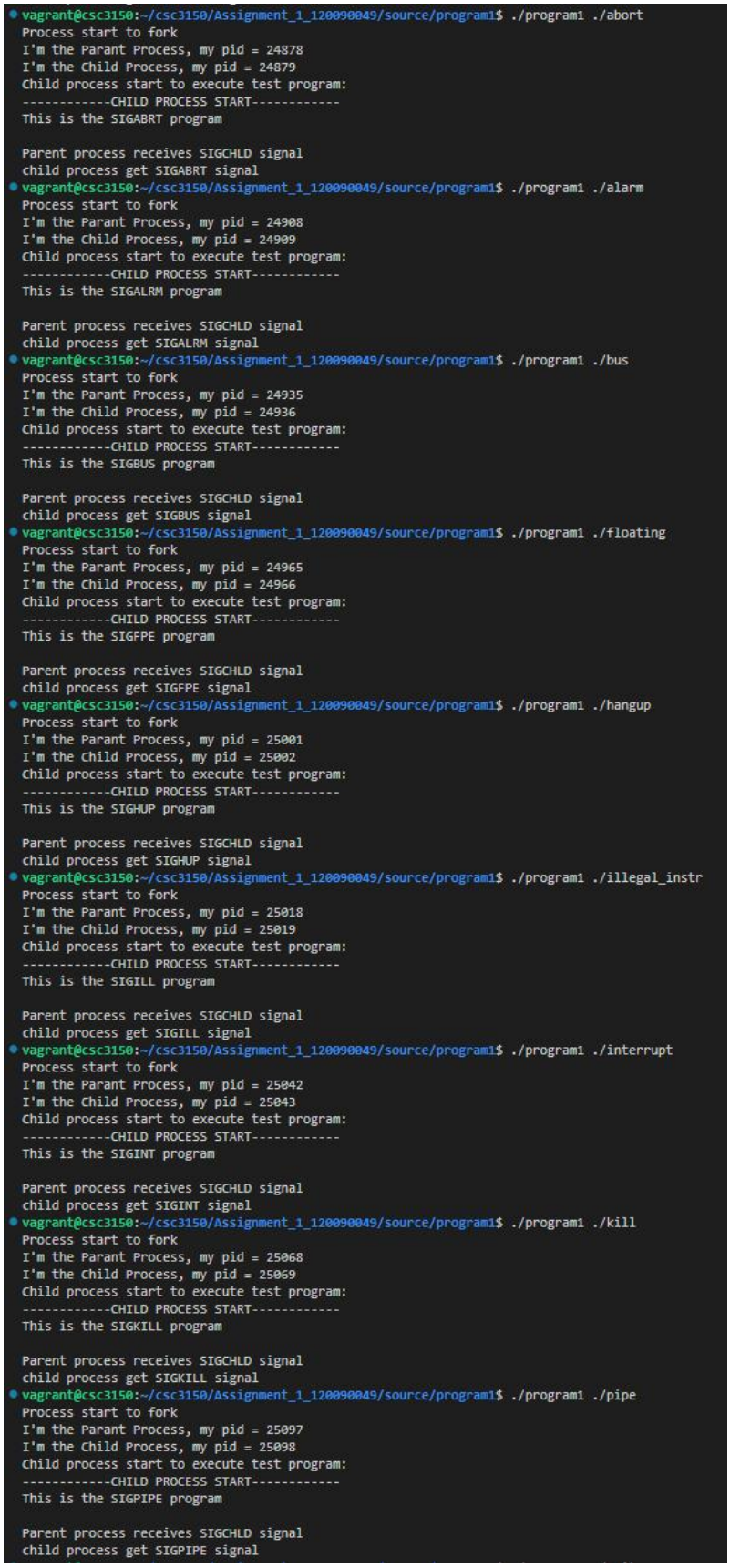


Fig 2. The output for the terminating signal

```
vagrant@csc3150:~/csc3150/Assignment_1_120090049/source/program1$ ./program1 ./quit
Process start to fork
I'm the Parant Process, my pid = 25120
I'm the Child Process, my pid = 25121
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGQUIT program

Parent process receives SIGCHLD signal
child process get SIGQUIT signal
vagrant@csc3150:~/csc3150/Assignment_1_120090049/source/program1$ ./program1 ./segment_fault
Process start to fork
I'm the Parant Process, my pid = 25147
I'm the Child Process, my pid = 25148
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGSEGV program

Parent process receives SIGCHLD signal
child process get SIGSEGV signal
vagrant@csc3150:~/csc3150/Assignment_1_120090049/source/program1$ ./program1 ./terminate
Process start to fork
I'm the Parant Process, my pid = 25165
I'm the Child Process, my pid = 25166
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGTERM program

Parent process receives SIGCHLD signal
child process get SIGTERM signal
vagrant@csc3150:~/csc3150/Assignment_1_120090049/source/program1$ ./program1 ./trap
Process start to fork
I'm the Parant Process, my pid = 25179
I'm the Child Process, my pid = 25180
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
```

Fig 3. The output for the terminating signal

2. Program 2

a) Program design

As is shown in figure 1, the steps of the program can be mainly divided into four parts.

i. Initialize the module, and create a kernel thread.

ii. Within the thread, fork a process to execute the test program in the child process.

iii. The parent process will wait until the child process terminates.

iv. Catch the signal from the test program and print it out in the kernel log.

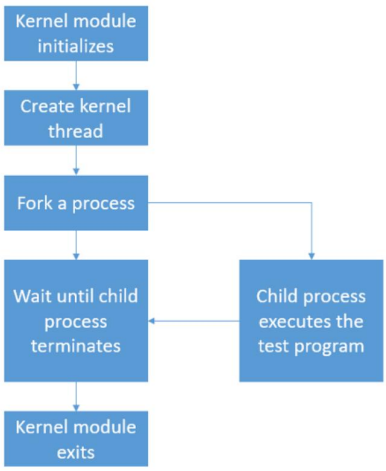


Fig 2. The main flow chart for program 2.

b) Detailed implementation

i. Export four functions from the kernel program, and recompile the kernel

```
extern pid_t kernel_clone(struct kernel_clone_args *args); // kernel/fork.c
extern int do_execve(struct filename *filename,
                     const char __user *const __user *__argv,
                     const char __user *const __user *__envp);
extern struct filename *getname_kernel(const char *filename);
extern long do_wait(struct wait_opts *wo);
```

ii. Initialize the module and create a kernel thread to run myfork function.

```
static int __init program2_init(void)
{
    // STEP 1 kernel module initializes
    printk("[program2] : module_init Lingpeng Chen 120090049\n");
    printk("[program2] : module_init create kthread start\n");

    // STEP 2 create kernel thread
    /* create a kernel thread to run my_fork */
    struct task_struct *myThread;
    myThread = kthread_create(&my_fork, NULL, "MyThread");
    // wake up new thread if ok
    if (!IS_ERR(myThread)) {
        printk("[program2] : module_init kthread start\n");
        wake_up_process(myThread);
    }

    return 0;
}
```

iii. In the myfork function, first initialize the k_sigaction, the data structure related to signal handling. The handler can keep track of the shared pending signals.

```
// initialize the date structure related to signal handling
// the handler keeps track of the shared pending signals
struct k_sigaction *ksa_handler = &current->sighand->action[0];
for (int k = 0; k < _NSIG; k++) {
    ksa_handler->sa.sa_handler = SIG_DFL;
    ksa_handler->sa.sa_flags = 0;
    ksa_handler->sa.sa_restorer = NULL;
    sigemptyset(&ksa_handler->sa.sa_mask);
    ksa_handler++;
}
```

iv. Within myfork, fork a process to execute the myexe function.

```
// STEP 3 Fork a process
/* fork a process using kernel_clone or kernel_thread */
// initialize the kernel clone arguments
struct kernel_clone_args kc_args = {
    // include in /include/linux/sched/task
    .flags =
        SIGCHLD, // How to clone the child process. When executing f
    .stack =
        (unsigned long)&myexe, // Specifies the location of the stac
    .stack_size = 0, //Normally set as 0 because it is unused.
    .parent_tid =
        NULL, // Used for clone() to point to user space memory in p
    .child_tid =
        NULL, // Used for clone() to point to user space memory in c
    .tls = 0, // Set thread local storage.
    .exit_signal = SIGCHLD,
};

/* execute a test program in child process */
// go to STEP 4 (child process executes the test program)
pid_t pid = kernel_clone(
    &kc_args); // Fork successfully: pid of child process
printk("[program2] : The child process has pid = %d\n", pid);
printk("[program2] : This is the parent process, pid = %d\n",
       (int)current->pid);
```

v. In the myexe function, we first have a sleep so that the kernel print of the PID and PPID can be executed in advance. Then, load and execute the executable file with the getname_kernel function and do_execve function, respectively.

```
// STEP 4 child process executes the test program
int myexe()
{
    msleep(1);
    int output;
    // char *file_path =
    //  "/home/vagrant/csc3150/Assignment_1_120090049/source/program2/test";
    char *file_path = "/tmp/test"; // pointer of the file path
    struct filename *my_file_name = getname_kernel(file_path);
    printk("[program2] : child process\n");
    output = do_execve(my_file_name, NULL, NULL);
    return 0;
}
```

vi. In the mywait function, the parent process will wait until the child process terminates. We set woflags as WUNTRACED | WEXITED in order to catch the stop or other terminating signal from the test program. Then we use do_wait() to wait for the child process. The signal from the child process will be stored in the "wo.wo_stat". Therefore, we can use a switch case statement to determine the received signal. Finally, we print them out in the kernel log.

```
// STEP 5 wait until child process terminates
mywait(pid);
```

```
void mywait(pid_t pid)
{
    static int status;
    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 = WUNTRACED | WEXITED;
    wo.wo_info = NULL;
    wo.wo_stat = (int __user *)&status;
    wo.wo_rusage = NULL;

    int a;
    a = do_wait(&wo);

    switch (wo.wo_stat) {···

    printk("[program2] : child process terminated\n");
    printk("[program2] : The return signal is %d\n", (wo.wo_stat));

    return;
}
```

c) Environment development

i. We need to modify the source code of the kernel so that we can export the function getname_kernel, kernel_thread, do_wait and do_exec.

```
2518
2519        return kernel_clone(&args);
2520    }
2521
2522    EXPORT_SYMBOL(kernel_thread);
2523
```

ii. Compile the kernel again. Then, insert the program.ko, wait for a few seconds and finally remove it. We can use the command “dmesg” to check the message of the kernel log.

d) Result

The output is listed as blow:

i. SIGABRT

```
[44253.270938] [program2] : module_init Lingpeng Chen 120090049
[44253.274766] [program2] : module_init create kthread start
[44253.278839] [program2] : module_init kthread start
[44253.282178] [program2] : The child process has pid = 29259
[44253.286666] [program2] : This is the parent process, pid = 29258
[44253.294637] [program2] : child process
[44253.553838] [program2] : get SIGTERM signal
[44253.557019] [program2] : child process terminated
[44253.560462] [program2] : The return signal is 134
[44255.302338] [program2] : module_exit./my
```

ii. SIGALARM

```
[44762.873102] [program2] : module_init Lingpeng Chen 120090049
[44762.876275] [program2] : module_init create kthread start
[44762.879404] [program2] : module_init kthread start
[44762.881917] [program2] : The child process has pid = 29335
[44762.885635] [program2] : This is the parent process, pid = 29334
[44762.894032] [program2] : child process
[44764.898299] [program2] : get SIGALARM signal
[44764.901037] [program2] : child process terminated
[44764.905289] [program2] : The return signal is 14
[44764.908720] [program2] : module_exit./my
```

iii. SIGBUS

```
[44833.833193] [program2] : module_init Lingpeng Chen 120090049
[44833.837145] [program2] : module_init create kthread start
[44833.842237] [program2] : module_init kthread start
[44833.846020] [program2] : The child process has pid = 29406
[44833.850502] [program2] : This is the parent process, pid = 29405
[44833.858275] [program2] : child process
[44834.126597] [program2] : get SIGBUS signal
[44834.130440] [program2] : child process terminated
[44834.135195] [program2] : The return signal is 135
[44835.865065] [program2] : module_exit./my
```

iv. SIGFPE

```
[44993.202334] [program2] : module_init Lingpeng Chen 120090049
[44993.206797] [program2] : module_init create kthread start
[44993.210777] [program2] : module_init kthread start
[44993.213997] [program2] : The child process has pid = 29483
[44993.218255] [program2] : This is the parent process, pid = 29482
[44993.226210] [program2] : child process
[44993.482945] [program2] : get SIGFPE signal
[44993.486148] [program2] : child process terminated
[44993.489732] [program2] : The return signal is 136
[44995.254644] [program2] : module_exit./my
```

v. SIGHUP

```
[45034.201422] [program2] : module_init Lingpeng Chen 120090049
[45034.205511] [program2] : module_init create kthread start
[45034.210038] [program2] : module_init kthread start
[45034.213821] [program2] : The child process has pid = 29554
[45034.218544] [program2] : This is the parent process, pid = 29553
[45034.225789] [program2] : child process
[45034.229102] [program2] : get SIGHUP signal
[45034.231774] [program2] : child process terminated
[45034.235695] [program2] : The return signal is 1
[45036.240633] [program2] : module_exit./my
```

vi. SIGILL

```
[45074.948737] [program2] : module_init Lingpeng Chen 120090049
[45074.952582] [program2] : module_init create kthread start
[45074.956039] [program2] : module_init kthread start
[45074.959014] [program2] : The child process has pid = 29628
[45074.963344] [program2] : This is the parent process, pid = 29627
[45074.974003] [program2] : child process
[45075.269213] [program2] : get SIGILL signal
[45075.272613] [program2] : child process terminated
[45075.276542] [program2] : The return signal is 132
[45076.977081] [program2] : module_exit./my
```

vii. SIGINT

```
[45118.759800] [program2] : module_init Lingpeng Chen 120090049
[45118.762994] [program2] : module_init create kthread start
[45118.766329] [program2] : module_init kthread start
[45118.768849] [program2] : The child process has pid = 29701
[45118.773357] [program2] : This is the parent process, pid = 29700
[45118.781777] [program2] : child process
[45118.785252] [program2] : get SIGINT signal
[45118.788044] [program2] : child process terminated
[45118.791234] [program2] : The return signal is 2
[45120.795405] [program2] : module_exit./my
```

viii. SIGKILL

```
[45147.498652] [program2] : module_init Lingpeng Chen 120090049
[45147.503460] [program2] : module_init create kthread start
[45147.507932] [program2] : module_init kthread start
[45147.511948] [program2] : The child process has pid = 29774
[45147.516588] [program2] : This is the parent process, pid = 29773
[45147.521596] [program2] : child process
[45147.524875] [program2] : get SIGKILL signal
[45147.528384] [program2] : child process terminated
[45147.532234] [program2] : The return signal is 9
[45149.528323] [program2] : module_exit./my
```

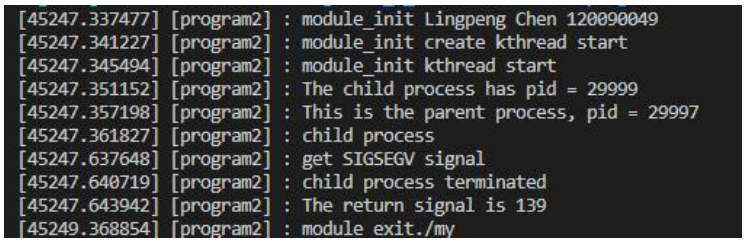
ix. SIGPIPE

```
[45178.290184] [program2] : module_init Lingpeng Chen 120090049
[45178.293800] [program2] : module_init create kthread start
[45178.298234] [program2] : module_init kthread start
[45178.302951] [program2] : The child process has pid = 29854
[45178.309068] [program2] : This is the parent process, pid = 29853
[45178.313936] [program2] : child process
[45178.318118] [program2] : get SIGPIPE signal
[45178.320952] [program2] : child process terminated
[45178.326517] [program2] : The return signal is 13
[45180.320674] [program2] : module_exit./my
```

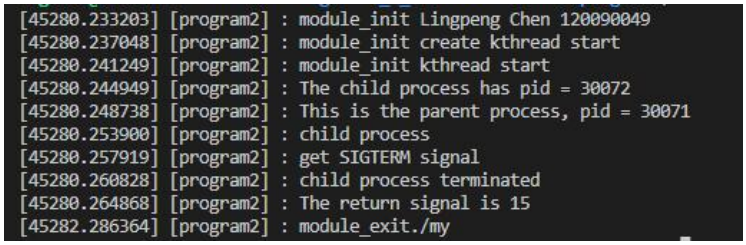
x. SIGQUIT

```
[45208.167250] [program2] : module_init Lingpeng Chen 120090049
[45208.171103] [program2] : module_init create kthread start
[45208.175156] [program2] : module_init kthread start
[45208.179521] [program2] : The child process has pid = 29925
[45208.183562] [program2] : This is the parent process, pid = 29924
[45208.189753] [program2] : child process
[45208.461046] [program2] : get SIGQUIT signal
[45208.463974] [program2] : child process terminated
[45208.467243] [program2] : The return signal is 131
[45210.201124] [program2] : module_exit./my
```

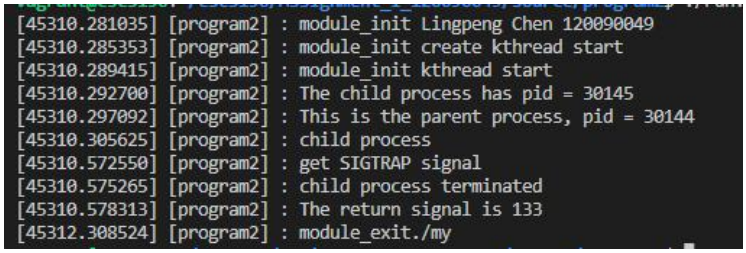
xi. SIGSEGV



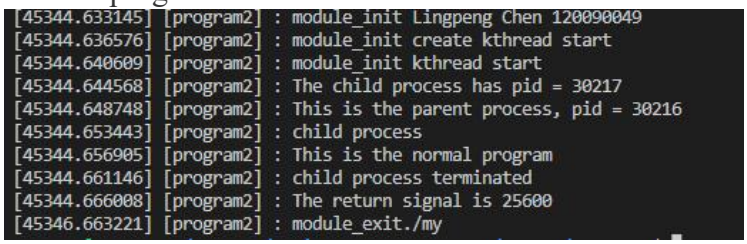
xii. SIGTERM



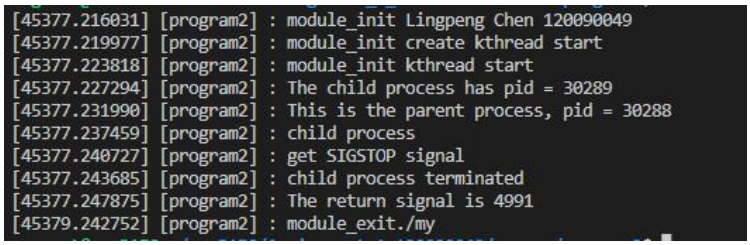
xiii. SIGTRAP



xiv. Normal program



xv. SIGSTOP



3. Bonus

a) Program design

In the bonus part, we are asked to implement the pstree function. The PID is located in the /proc folder. We can have access to the detailed information of the PID by entering “more status” command (As is shown in figure 4). From the picture, we can get PID, name, PPID. Then, we can traverse through all folders. Then we can get all processes. Also, we can get into the task folder to check the threads of the process (As is shown in figure 5).

Then, based on the information, we can create a tree. Furthermore, we can use this tree to print out the pstree.

```
vagrant@csc3150:/proc$ cd 1
vagrant@csc3150:/proc/1$ more status
Name:   systemd
Umask:  0000
State:  S (sleeping)
Tgid:   1
Ngid:   0
Pid:    1
PPid:   0
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
```

Fig 4. Command result.

```
vagrant@csc3150:/proc/15337$ cd task
vagrant@csc3150:/proc/15337/task$ ls
15337  15338  15339  15340  15341  15342  15343  15344  15345  15346  15347  15348  15349  15350  15351  28352
```

Fig 5. Command result.

b) Detailed implementation

i. Traverse all the folders under the /proc folder. Then, store the information of the processes and threads by using the struct “pid_ppid.” At this moment, we can only fill in the PID, PPID, and the PID name. And all “pid_ppid” will be stored in a vector called process_list.

```
struct pid_ppid {
    int pid;
    int ppid;
    string name;
    struct pid_ppid *parent;
    vector<pid_ppid *> sons;
    // char status[8];
};
```

ii. To build a tree, traverse the vector of process_list and fill in the parent and sons of each element.

iii. Use the recursive method to print the tree. During the implementation, I used “+” to represent an intersect point. And I use “-” and “|” as lines.

c) Result

In total, I realize three functions.

i. pstree -V

```
vagrant@csc3150:~/csc3150/Assignment_1_120090049/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.
```

ii. pstree -c

```
+--systemd+--systemd-journal
          +--lvmetad
          +--systemd-udevd
          +--cpptools-srv+--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          |              +--{cpptools-srv}
          +--dhclient
          +--acpid
          +--iscsid
          +--iscsid
          +--accounts-daemon+--{gmain}
          |                 +--{gdbus}
          +--cron
          +--dbus-daemon
          +--lxcfs+--{lxcfs}
          |       +--{lxcfs}
          |       +--{lxcfs}
          +--systemd-logind
          +--atd
          +--rsyslogd+--{in:imuxsock}
          |          +--{in:imklog}
          |          +--{rs:main}
          +--polkitd+--{gmain}
          |         +--{gdbus}
          +--sshd+--sshd+--sshd+--bash+--sh+--node+--node+--bash
          |      |      |      |      |   |      |      +--bash+--pstree
          |      |      |      |      |   |      |      +--bash
          |      |      |      |      |   |      +--node+--cpptools+--{cpptools}
```

iii. pstree -p

```
+--systemd(1)+--systemd-journal(430)
             +--lvmetad(441)
             +--systemd-udevd(464)
             +--cpptools-srv(813)+--{cpptools-srv}(814)
             |                   +--{cpptools-srv}(815)
             |                   +--{cpptools-srv}(816)
             |                   +--{cpptools-srv}(817)
             |                   +--{cpptools-srv}(818)
             |                   +--{cpptools-srv}(819)
             |                   +--{cpptools-srv}(820)
             |                   +--{cpptools-srv}(821)
             |                   +--{cpptools-srv}(822)
             |                   +--{cpptools-srv}(823)
             |                   +--{cpptools-srv}(824)
             |                   +--{cpptools-srv}(825)
             |                   +--{cpptools-srv}(826)
             |                   +--{cpptools-srv}(827)
             |                   +--{cpptools-srv}(828)
             +--dhclient(875)
             +--acpid(1031)
             +--iscsid(1032)
             +--iscsid(1033)
             +--accounts-daemon(1036)+--{gmain}(1045)
             |                       +--{gdbus}(1047)
             +--cron(1040)
             +--dbus-daemon(1042)
             +--lxcfs(1049)+--{lxcfs}(1055)
             |             +--{lxcfs}(1056)
             |             +--{lxcfs}(24090)
             +--systemd-logind(1053)
             +--atd(1058)
             +--rsyslogd(1059)+--{in:imuxsock}(1066)
             |                +--{in:imklog}(1067)
             |                +--{rs:main}(1068)
             +--polkitd(1079)+--{gmain}(1089)
             |               +--{gdbus}(1093)
             +--sshd(1094)+--sshd(1594)+--sshd(1686)+--bash(1688)+--sh(1758)+--node(1768)+--node(1848)+--bash(2497)
             |            |           |             |             |          |           |           +--bash(11955)+--pstree(882)
             |            |           |             |             |          |           |           +--bash(28971)
             |            |           |             |             |          |           +--node(1881)+--cpptools(2118)+--{cpptools}(2119)
             |            |           |             |             |          |           |           |                    +--{cpptools}(2120)
             |            |           |             |             |          |           |           |                    +--{cpptools}(2121)
             |            |           |             |             |          |           |           |                    +--{cpptools}(2122)
             |            |           |             |             |          |           |           |                    +--{cpptools}(2123)
             |            |           |             |             |          |           |           |                    +--{cpptools}(2124)
             |            |           |             |             |          |           |           |                    +--{cpptools}(2125)
             |            |           |             |             |          |           |           |                    +--{cpptools}(2126)
```

4. Conclusion

In this assignment, I learned how to create and execute the process in the user mode or kernel mode. And I also learned to catch and analyze the signal raised by the process. In the user mode, we can directly use functions like execve or waitpid. These functions execute the system calls in the kernel mode and return to the user mode. In the kernel mode, however, we have to export the kernel function by ourselves. We need to load and remove the LKM to run the kernel program. This whole process helps me better understand how the kernel works.

In the bonus part, I learned how to have access to detailed information about processes and threads. And I also improve my programming skills by designing how to print a tree.

In conclusion, this assignment helped me better understand the kernel of the Linux system and improved my programming skills.