

Operating System

Kernel-Mode Multi-Process Programming

10/8/2022

CONTENTS

1. Development environment set-up

1.1 Remote SSH

1.2 Compile Kernel

2. Program design & Output

2.1 Task 1

2.2 Task 2

2.3 Bonus

3. Learning outcome

1. Development environment set-up

1.1 Remote SSH

The development environment is established with VSCode and VMware. The host machine is running Window 11 22H2 and the virtual machine is running Ubuntu 18.04 (downloaded from <https://releases.ubuntu.com/18.04>) with Linux kernel version 4.15 (before recompilation).

In order to establish a SSH remote connect from the host machine to the virtual machine, I first use

```
sudo apt-get install openssh-server
```

command in Linux terminal to install openssh on the VM, and then install the Remote – SSH extension for the VSC on the host machine.

After that, config a new SSH connection with

```
ssh username@hostname
```

in the prompted command palette in VSC and wait for the connection initialization. Then, enter the VM user password access the VM in the host machine VSC.

Additionally, some essential extensions & tools such as should be reinstalled remotely. Specifically, install C/C++ extension on the VM, also install GCC via

```
sudo apt-get install update  
sudo apt-get install build-essential gdb
```

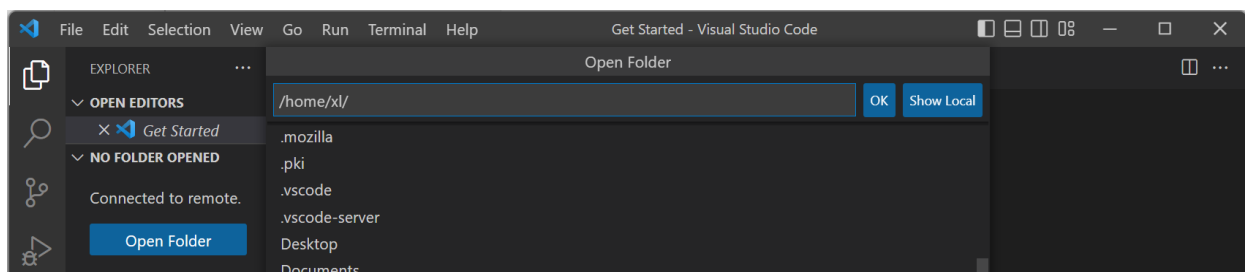


Figure 1: SSH Connected

1.2 Compile Kernel

Since in task 2 we need to write a Loadable Kernel Module (LKM) with several functions exported from the Linux Kernel with version 5.10.x, we need to rebuild the current kernel.

The basic procedure is download kernel -> modify source code (export symbols) -> recompile kernel -> install kernel.

The detailed procedures are given below:

System Config: CPU: 8 Memory: 8GB Disk: 50GB

1. Download source from www.kernel.com (.xyz) (**v5.10.145**)
2. Copy this file to the designated workplace (/home/workplace)
3. Extract file here
4. Modify the source code by EXPORT_SYMBOL ()

Note that this function should be used below the #include declarations and by convention just right below the target function implementation

In my case, four functions are exported, namely:

- 1) **do_wait** (/kernel/exit.c)

```
EXPORT_SYMBOL(do_wait);
```

- 2) **kernel_clone** (/kernel/fork.c)

```
EXPORT_SYMBOL(kernel_clone);
```

- 3) **getname_kernel** (/fs/namei.c)

```
EXPORT_SYMBOL(getname_kernel);
```

- 4) **do_execve** (/fs/exec.c)

```
EXPORT_SYMBOL(do_execve);
```

5. Install dependency

```
sudo apt-get install libncurses-dev gawk flex bison openssl libssl-dev dkms libelf-dev libudev-dev libpci-dev libiberty-dev autoconf llvm dwarves
```

6. Copy the current kernel config file to the workplace and rename it as .config

```
cp -v /boot/config-$(uname -r) .config
```

7. Clean & Load config

```
make mrproper
```

```
make clean
```

```
make menuconfig (Load the .config file, save and exit)
```

8. Compile (TAKES UP MOST OF THE TIME)

```
make -j$(nproc)
```

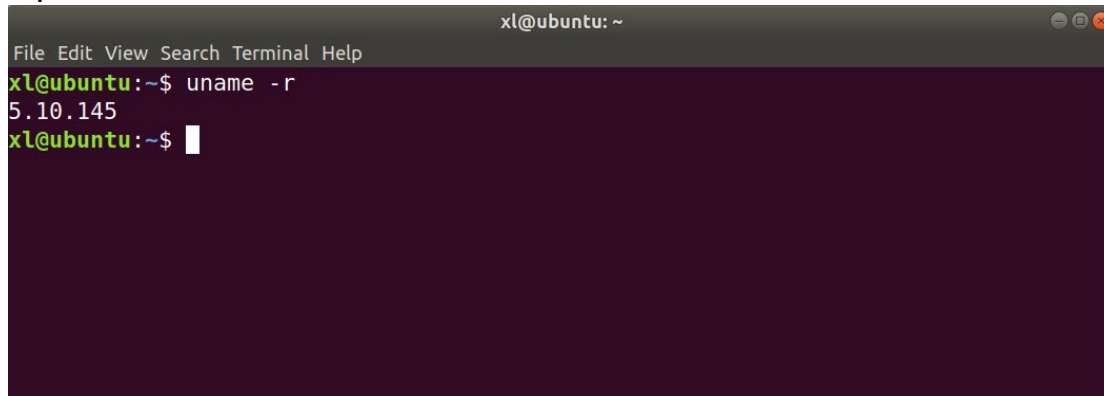
9. Install

```
sudo make modules_install  
sudo make install
```

10. Reboot

```
Reboot
```

Upon success, use `uname -r` to check the kernel version:

A terminal window titled 'xl@ubuntu: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'xl@ubuntu:~\$'. The command 'uname -r' has been entered, and the output '5.10.145' is displayed. The prompt is now 'xl@ubuntu:~\$' with a cursor.

```
xl@ubuntu:~$ uname -r  
5.10.145  
xl@ubuntu:~$
```

Figure 2: Check Kernel Version

2. Program Design

2.1 Task 1

In this task, the program runs a process in the user mode and forks a child process to execute the task program while the parent process wait until the child process terminates. Then the program will print out the corresponding exit status message and signal information.

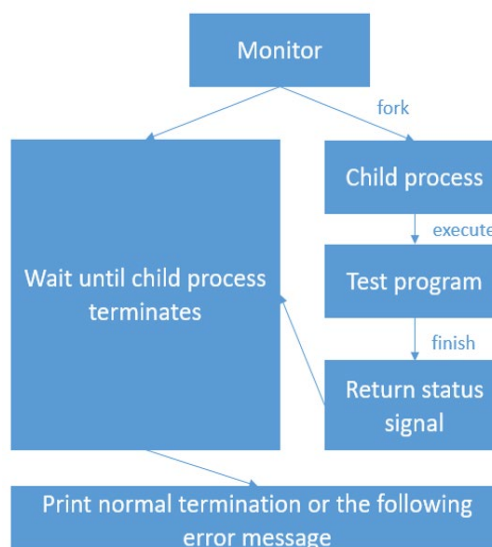


Figure 3: flow chart for program1

To implement, use `fork()` in the main function to fork a child

process, and do the if-else branch for the parent and child process respectively **based on the return value of the fork()**.

In the parent process where the return value is child pid (>0), use waitpid() to wait the child process to terminate.

```
/* wait for child process terminates */  
// waitpid returns the child pid  
rt_pid = waitpid(pid, &chld_status, WUNTRACED);
```

Figure 4: waitpid()

Note that WUNTRACED here is to make the process return to the parent when stopped. When returns to the parent process, the parent program receives SIGCHLD information (demonstrate by printing) and prints out the signal information based on the child status. Three conditions are considered here: normal termination, signaled exit, and stopped. For the signaled exit condition, there are **two predefined char pointer arrays** following the correct sequence (the signal number) for the output.

```
const char *POSIX_SIG[] = { NULL, "SIGHUP", "SIGINT", "SIGQUIT",  
                                "SIGILL", "SIGTRAP", "SIGABRT", "SIGBUS",  
                                "SIGFPE", "SIGKILL", NULL, "SIGSEGV",  
                                NULL, "SIGPIPE", "SIGALRM", "SIGTERM" };  
  
const char *SIG_NAME[] = { NULL,  
                            "hangup",  
                            "interrupt",  
                            "quit",  
                            "illegal instruction",  
                            "trap",  
                            "abort",  
                            "bus error",  
                            "floating-point exception",  
                            "kill",  
                            NULL,  
                            "segment fault",  
                            NULL,  
                            "pipe",  
                            "alarm",  
                            "terminate" };
```

Figure 5: signal information

In the child process where fork() returns 0, use execve() to execute the target file.

```
printf("Child process start to execute test program:\n");  
execve(arg[0], arg, NULL);
```

Figure 6: execve()

The outputs for the task 1 are given below

Please refer to the code for the final revision for the output!!

```
x1@ubuntu:~/Desktop/template_source/source/program1$ ./program1 normal
Process start to fork
I'm the Parent Process, my pid = 54535
I'm the Child Process, my pid = 54536
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
```

Figure 7: Task1-Normal

```
x1@ubuntu:~/Desktop/template_source/source/program1$ ./program1 stop
Process start to fork
I'm the Parent Process, my pid = 54562
I'm the Child Process, my pid = 54563
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
```

Figure 8: Task1- Stop

```
x1@ubuntu:~/Desktop/template_source/source/program1$ ./program1 abort
Process start to fork
I'm the Parent Process, my pid = 54591
I'm the Child Process, my pid = 54592
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
```

Figure 9: Task1-Abort

```

xl@ubuntu:~/Desktop/template_source/source/program1$ ./program1 alarm
Process start to fork
I'm the Parent Process, my pid = 54618
I'm the Child Process, my pid = 54619
Child process start to execute test program:
-----CHILD PROCESS START-----
This is the SIGALRM program

Parent process receives SIGCHLD signal
Child process get SIGALRM signal

```

Figure 10: Task1 – Alarm

```

xl@ubuntu:~/Desktop/template_source/source/program1$ ./program1 bus
Process start to fork
I'm the Parent Process, my pid = 54657
I'm the Child Process, my pid = 54658
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

```

Figure 11: Task1 – Bus

```

xl@ubuntu:~/Desktop/template_source/source/program1$ ./program1 floating
Process start to fork
I'm the Parent Process, my pid = 54685
I'm the Child Process, my pid = 54686
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

```

Figure 12: Task1 – floating

```

xl@ubuntu:~/Desktop/template_source/source/program1$ ./program1 hangup
Process start to fork
I'm the Parent Process, my pid = 54700
I'm the Child Process, my pid = 54701
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

```

Figure 13: Task1 – hangup


```

xl@ubuntu:~/Desktop/template_source/source/program1$ ./program1 illegal_instr
Process start to fork
I'm the Parent Process, my pid = 54739
I'm the Child Process, my pid = 54740
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

```

Figure 14: Task1 - illegal_instr

```

xl@ubuntu:~/Desktop/template_source/source/program1$ ./program1 interrupt
Process start to fork
I'm the Parent Process, my pid = 54794
I'm the Child Process, my pid = 54795
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

```

Figure 15: Task1 – interrupt

```

xl@ubuntu:~/Desktop/template_source/source/program1$ ./program1 kill
Process start to fork
I'm the Parent Process, my pid = 54808
I'm the Child Process, my pid = 54809
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

```

Figure 16: Task1 – kill

```

xl@ubuntu:~/Desktop/template_source/source/program1$ ./program1 pipe
Process start to fork
I'm the Parent Process, my pid = 54835
I'm the Child Process, my pid = 54836
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

```

Figure 17: Task1 – pipe

```

xl@ubuntu:~/Desktop/template_source/source/program1$ ./program1 quit
Process start to fork
I'm the Parent Process, my pid = 54861
I'm the Child Process, my pid = 54862
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

```

Figure 18: Task1 – quit

```

xl@ubuntu:~/Desktop/template_source/source/program1$ ./program1 segment_fault
Process start to fork
I'm the Parent Process, my pid = 54891
I'm the Child Process, my pid = 54892
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

```

Figure 19: Task1 - segment_fault

```

xl@ubuntu:~/Desktop/template_source/source/program1$ ./program1 terminate
Process start to fork
I'm the Parent Process, my pid = 54907
I'm the Child Process, my pid = 54908
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

```

Figure 20: Task1 – terminate

```

xl@ubuntu:~/Desktop/template_source/source/program1$ ./program1 trap
Process start to fork
I'm the Parent Process, my pid = 54934
I'm the Child Process, my pid = 54935
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

```

Figure 21: Task1 – trap

2.2 Task 2

In this task, the program2.ko (LKM) initiates by creating a kernel thread, in which the process will fork a child process and the parent process will wait for the child process to terminate and then print out the signal related information, just similar to what happens in program1.

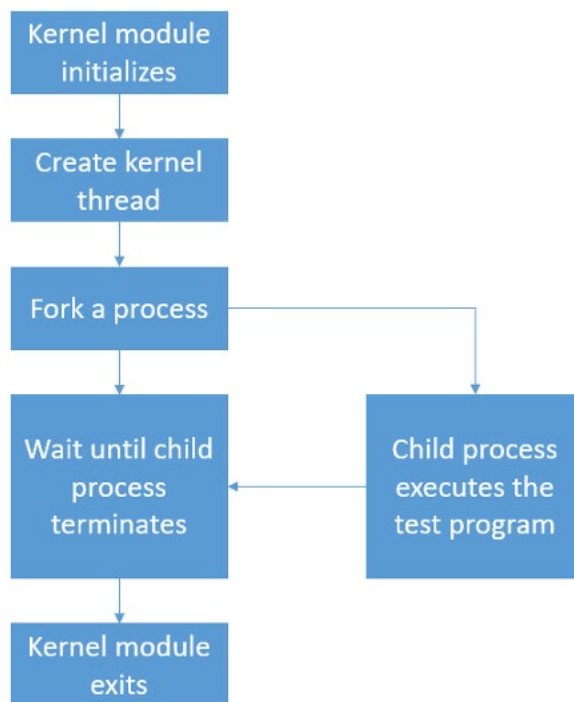


Figure 22: flow chart for task2

To implement, since we are writing a Loadable Kernel Module, we need to implement some functions we used in task 1 (in user mode) by ourselves. Four function symbols, including **do_wait()**, **kernel_clone()**, **getname_kernel()**, and **do_execve()** should be exported before and declared as extern functions.

```
// use exported functions
extern long do_wait(struct wait_opts *wo);

extern int do_execve(struct filename *filename,
                    const char __user *const __argv,
                    const char __user *const __envp);

extern pid_t kernel_clone(struct kernel_clone_args *kargs);

extern struct filename *getname_kernel(const char *filename);
```

Figure 23: Extern functions

When initiates, the program will call the **kthread_create()** function to create a kernel thread doing myfork() function. And pcs_info as the return object of the kthread_create() contains the essential process information of the created process. If there's no error in creating this process, the program will continue to do the what's contained in the myfork() function. If not, print the error info.

```
pcs_info = kthread_create(&my_fork, NULL, "My Thread");

if (!IS_ERR(pcs_info)) {
    printk("[program2] : Module_init kthread start\n");
    wake_up_process(pcs_info);
} else {
    PTR_ERR(pcs_info);
}
```

Figure 24: kthread_create

In **myfork()**, the program will fork a child process by invoking **kernel_clone()**, and this child process will do the **myexec()** function while the parent process will wait the child process to terminate by executing **mywait()**.

```
// set kernel_clone_args parameters
kargs.flags = SIGCHLD;
kargs.stack =
    (unsigned long)&my_exec; // let the child process execute my_exec function
kargs.stack_size = 0;
kargs.parent_tid = NULL;
kargs.child_tid = NULL;
kargs.tls = 0;
kargs.exit_signal = SIGCHLD;

/* fork a process using kernel_clone or kernel_thread */
/* execute a test program in child process */

chld_pid = kernel_clone(&kargs);
```

Figure 25: kernel_clone

In **myexec()**, the child process will execute the designated task file using **getname_kernel()** to get the filename and **do_execve()** to execute that executable file.

Note that do_execve will return 0 (NULL) upon success.

```
fopen = getname_kernel(path);
ret = do_execve(fopen, NULL, NULL);
```

Figure 26: myexec

In **mywait()**, the parent process will wait the child process to terminate or stop using **do_wait()** function.

```
struct wait_opts wo = { .wo_type = PIDTYPE_PID,
                        .wo_pid = find_get_pid(pid),
                        .wo_info = NULL,
                        .wo_flags = WEXITED | WUNTRACED,
                        .wo_stat = status,
                        .wo_rusage = NULL };
```

Figure 27: wait_opts parameter

After the child process has executed the target file, the exit signal will be passed to the parent process via **wait_opts.wo_stat** (an int). So after returns to the parent process in **myfork()**, **signal_display()** is structured to handle this signal and print the corresponding information on the kernel log.

Note that the signal number need to do a simple arithmetic operation to match the POSIX signal numbers.

```
int status_display(int e_status)
{
    int r_status = e_status % 128;
    // normal termination
    if (e_status == 0) {
        printk("[program2] : Child process get normal termination");
        return 0;
    }
    // stop
    else if (r_status == 127) {
        printk("[program2] : Get SIGSTOP signal");
        return 19;
    } else {
        // hang up
        if (r_status == 1) {
            printk("[program2] : Get SIGHUP signal");
        }
        // interrupt
        else if (r_status == 2) {
            printk("[program2] : Get SIGINT signal");
        }
    }
}
```

Figure 28: signal_display

The outputs for the task 2 are given below

Please refer to the code for the final revision for the output!!

```
[113597.809159] [program2] : Module_init {LiuXiaoyuan} {120040051}
[113597.809163] [program2] : Module_init create kthread start
[113597.809251] [program2] : Module_init kthread start
[113597.809303] [program2] : The child process has pid = 54383
[113597.809332] [program2] : This is the parent process, pid = 54382
[113597.809333] [program2] : Child process
[113597.810418] [program2] : Child process get normal termination with EXIT STATUS = 0
[113601.327747] [program2] : Module_exit
```

Figure 29: Task2 – normal

```
[113126.254839] [program2] : Module_init {LiuXiaoyuan} {120040051}
[113126.254841] [program2] : Module_init create kthread start
[113126.254984] [program2] : Module_init kthread start
[113126.255044] [program2] : The child process has pid = 52464
[113126.255045] [program2] : This is the parent process, pid = 52463
[113126.255055] [program2] : Child process
[113126.255952] [program2] : Get SIGSTOP signal
[113126.255954] [program2] : Child process terminated
[113126.255955] [program2] : The return signal is 19
[113132.871774] [program2] : Module_exit
```

Figure 30: Task2 – stop

```
[113071.040400] [program2] : Module_init {LiuXiaoyuan} {120040051}
[113071.040403] [program2] : Module_init create kthread start
[113071.040505] [program2] : Module_init kthread start
[113071.040574] [program2] : The child process has pid = 52012
[113071.040576] [program2] : This is the parent process, pid = 52011
[113071.040582] [program2] : Child process
[113071.135466] [program2] : Get SIGABRT signal
[113071.135469] [program2] : Child process terminated
[113071.135470] [program2] : The return signal is 6
[113073.254713] [program2] : Module_exit
```

Figure 31: Task2 – abort

```
[113022.567503] [program2] : Module_init {LiuXiaoyuan} {120040051}
[113022.567506] [program2] : Module_init create kthread start
[113022.567605] [program2] : Module_init kthread start
[113022.567659] [program2] : The child process has pid = 51567
[113022.567661] [program2] : This is the parent process, pid = 51566
[113022.567667] [program2] : Child process
[113024.568868] [program2] : Get SIGALRM signal
[113024.568870] [program2] : Child process terminated
[113024.568872] [program2] : The return signal is 14
[113025.285784] [program2] : Module_exit
```

Figure 32: Task2 – alarm


```

[112979.182745] [program2] : Module_init {LiuXiaoyuan} {120040051}
[112979.182747] [program2] : Module_init create kthread start
[112979.182818] [program2] : Module_init kthread start
[112979.182872] [program2] : The child process has pid = 51153
[112979.182874] [program2] : This is the parent process, pid = 51152
[112979.182883] [program2] : Child process
[112979.275101] [program2] : Get SIGBUS signal
[112979.275104] [program2] : Child process terminated
[112979.275105] [program2] : The return signal is 7
[112981.405730] [program2] : Module_exit

```

Figure 33: Task2 - bus error

```

[112909.151027] [program2] : Module_init {LiuXiaoyuan} {120040051}
[112909.151030] [program2] : Module_init create kthread start
[112909.151127] [program2] : Module_init kthread start
[112909.151183] [program2] : The child process has pid = 50709
[112909.151185] [program2] : This is the parent process, pid = 50708
[112909.151193] [program2] : Child process
[112909.246803] [program2] : Get SIGFPE signal
[112909.246805] [program2] : Child process terminated
[112909.246806] [program2] : The return signal is 8
[112911.038872] [program2] : Module_exit

```

Figure 34: Task2 - floating point exception

```

[112872.630682] [program2] : Module_init {LiuXiaoyuan} {120040051}
[112872.630684] [program2] : Module_init create kthread start
[112872.630842] [program2] : Module_init kthread start
[112872.630897] [program2] : The child process has pid = 50272
[112872.630898] [program2] : This is the parent process, pid = 50271
[112872.630925] [program2] : Child process
[112872.632105] [program2] : Get SIGHUP signal
[112872.632107] [program2] : Child process terminated
[112872.632108] [program2] : The return signal is 1
[112875.364626] [program2] : Module_exit

```

Figure 35: Task2 - hangup

```

[112825.307500] [program2] : Module_init {LiuXiaoyuan} {120040051}
[112825.307518] [program2] : Module_init create kthread start
[112825.307687] [program2] : Module_init kthread start
[112825.307745] [program2] : The child process has pid = 49794
[112825.307746] [program2] : This is the parent process, pid = 49793
[112825.307759] [program2] : Child process
[112825.407262] [program2] : Get SIGILL signal
[112825.407264] [program2] : Child process terminated
[112825.407266] [program2] : The return signal is 4
[112827.812979] [program2] : Module_exit

```

Figure 36: Task2 - illegal instruction

```

[112771.406523] [program2] : Module_init {LiuXiaoyuan} {120040051}
[112771.406525] [program2] : Module_init create kthread start
[112771.406653] [program2] : Module_init kthread start
[112771.406723] [program2] : The child process has pid = 49359
[112771.406724] [program2] : This is the parent process, pid = 49358
[112771.406731] [program2] : Child process
[112771.407929] [program2] : Get SIGINT signal
[112771.407930] [program2] : Child process terminated
[112771.407932] [program2] : The return signal is 2
[112773.621781] [program2] : Module_exit

```

Figure 37: Task2 – interrupt

```

[112700.701714] [program2] : Module_init {LiuXiaoyuan} {120040051}
[112700.701717] [program2] : Module_init create kthread start
[112700.701793] [program2] : Module_init kthread start
[112700.701847] [program2] : The child process has pid = 48943
[112700.701849] [program2] : This is the parent process, pid = 48942
[112700.701858] [program2] : Child process
[112700.702856] [program2] : Get SIGKILL signal
[112700.702859] [program2] : Child process terminated
[112700.702861] [program2] : The return signal is 9
[112703.829717] [program2] : Module_exit

```

Figure 38: Task2 – kill

```

[112642.206246] [program2] : Module_init {LiuXiaoyuan} {120040051}
[112642.206249] [program2] : Module_init create kthread start
[112642.206306] [program2] : Module_init kthread start
[112642.206360] [program2] : The child process has pid = 48504
[112642.206362] [program2] : This is the parent process, pid = 48503
[112642.206369] [program2] : Child process
[112642.207318] [program2] : Get SIGPIPE signal
[112642.207319] [program2] : Child process terminated
[112642.207320] [program2] : The return signal is 13
[112644.534123] [program2] : Module_exit

```

Figure 39: Task2 – pipe

```

[112591.678510] [program2] : Module_init {LiuXiaoyuan} {120040051}
[112591.678512] [program2] : Module_init create kthread start
[112591.678606] [program2] : Module_init kthread start
[112591.678658] [program2] : The child process has pid = 48067
[112591.678660] [program2] : This is the parent process, pid = 48066
[112591.678664] [program2] : Child process
[112591.772254] [program2] : Get SIGQUIT signal
[112591.772256] [program2] : Child process terminated
[112591.772258] [program2] : The return signal is 3
[112594.948775] [program2] : Module_exit

```

Figure 40: Task2 – quit


```
[112498.350670] [program2] : Module_init {LiuXiaoyuan} {120040051}
[112498.350673] [program2] : Module_init create kthread start
[112498.350890] [program2] : Module_init kthread start
[112498.350962] [program2] : The child process has pid = 47608
[112498.350964] [program2] : This is the parent process, pid = 47607
[112498.350971] [program2] : Child process
[112498.449614] [program2] : Get SIGSEGV signal
[112498.449616] [program2] : Child process terminated
[112498.449618] [program2] : The return signal is 11
[112500.285565] [program2] : Module_exit
```

Figure 41: Task2 - segment fault

```
[112406.054458] [program2] : Module_init {LiuXiaoyuan} {120040051}
[112406.054461] [program2] : Module_init create kthread start
[112406.066976] [program2] : Module_init kthread start
[112406.067053] [program2] : The child process has pid = 47108
[112406.067055] [program2] : This is the parent process, pid = 47107
[112406.067062] [program2] : Child process
[112406.068438] [program2] : Get SIGTERM signal
[112406.068440] [program2] : Child process terminated
[112406.068442] [program2] : The return signal is 15
[112409.452142] [program2] : Module_exit
```

Figure 42: Task2 – terminate

```
[112316.879148] [program2] : Module_init create kthread start
[112316.879443] [program2] : Module_init kthread start
[112316.879523] [program2] : The child process has pid = 46658
[112316.879525] [program2] : This is the parent process, pid = 46657
[112316.879536] [program2] : Child process
[112316.979345] [program2] : Get SIGTRAP signal
[112316.979348] [program2] : Child process terminated
[112316.979349] [program2] : The return signal is 5
[112320.005112] [program2] : Module_exit
```

Figure 43: Task2 – trap

2.3 Bonus

```
x1@ubuntu:~/Desktop/Assignment_1_120040051/source/bonus$ ./pstree -p
----- PRINT START -----
|-kthreadd (2)
  |-kworker/7 (88008)
  |-kworker/7 (88007)
  |-kworker/0 (87962)
  |-kworker/3 (87935)
  |-kworker/u256 (87717)
  |-kworker/0 (87601)
  |-kworker/2 (84393)
  |-kworker/6 (80197)
  |-kworker/4 (79945)
  |-kworker/5 (79942)
  |-kworker/1 (79916)
  |-kworker/u256 (79444)
  |-kworker/u256 (76620)
  |-kworker/2 (41977)
  |-kworker/4 (40730)
  |-kworker/7 (40727)
  |-kworker/5 (40690)
  |-kworker/3 (36950)
```

Figure 44: pstree -p

```
x1@ubuntu:~/Desktop/Assignment_1_120040051/source/bonus$ ./pstree -A
----- PRINT START -----
|-kthreadd
  |-kworker/7
  |-kworker/0
  |-kworker/3
  |-kworker/u256
  |-kworker/2
  |-kworker/6
  |-kworker/4
  |-kworker/5
  |-kworker/1
  |-kworker/u256
  |-kworker/u256
```

Figure 45: pstree -A

```

xl@ubuntu:~/Desktop/Assignment_1_120040051/source/bonus$ ./pstree -l
----- PRINT START -----
|-kthreadd
  |-kworker/7
  |-kworker/0
  |-kworker/3
  |-kworker/u256
  |-kworker/2
  |-kworker/6
  |-kworker/4
  |-kworker/5
  |-kworker/1
  |-kworker/u256
  |-kworker/u256
  |-kworker/2
  |-kworker/4

```

Figure 46: pstree -l

```

xl@ubuntu:~/Desktop/Assignment_1_120040051/source/bonus$ gcc -o pstree pstree.c
xl@ubuntu:~/Desktop/Assignment_1_120040051/source/bonus$ python ts.py /home/xl/Desktop/Assignment_1_120040051/source/bonus
test passed.

```

Figure 47: Passed Test

3. Learning outcome

Through this program, **I've learned:**

1. Build Remote SSH connection from host machine to virtual machine in VS Code
2. Read the modify (a little bit) Linux kernel source code
3. Compile and install the kernel from the source file using Linux command
4. The basic mechanism in Linux for process creation, file execution, signal handling.
5. Use `fork()`, `wait()/waitpid()`, `execve()` function to implement the above
6. Program a simple Loadable Kernel Module (LKM) in C
7. Insert and remove LKM, print out the kernel log with Linux command
8. Use `kernel_clone()`, `getname_kernel()`, `do_wait()`, and `do_execve()` to the similar things as 5 in kernel module
9. Use clang-format to format the code
10. Use `ps` command to track process and threads (**bonus**)
11. Get process status in `/proc` folder (**bonus**)