CSC 3150 Project #1

Ziyu XIE
121090642

October 7, 2023

Contents

1 Introduction

1.1 Project tasks

This project includes two tasks. The first one is to fork a child process and execute a test program. After that, we wait for the test program and get the signal. The second one asks us to create a kernel thread and run **`my_fork`** function. In this function, we should execute the test program, wait for it, and get the signal.

1.2 Development environment

The development environment is Ubuntu 16.04 with Linux kernel version 5.10.197. The gcc version is 5.4.0.

To set up this environment, we should first get the correct Linux kernel version installed. Follow the command lines below to get the kernel version 5.10.197.

```
sudo su # get the root permission
cd /home/seed/work # if not having the dir, create one
wget xxx # get the kernel source code, xxx represents the
    downloading link
tar xvf linux-5.10.197.tar.xz # unzip
cd /boot # get into boot dir
cp config* /home/seed/work/linux-5.10.197 # here * means there
    is still some chars after the config

cd /home/seed/work/linux-5.10.197
make mrproper
make clean
make menuconfig
# save and eixt
make # to build the kernel image and modules
make modules_install
make install
reboot # after reboot, the newest version should be 5.10.197
```

Till now, the environment is enough for program1. However, for program2, we need kernel functions, which means that we should modify the kernel to use them. Follow the instrucions below to recompile the kernel for program2 usage.

```
sudo su # get the root permission
chmod 777 -R /home/seed/work # help vscode get the permission
    to modify files
# then add the following lines in corresponding files
# EXPORT_SYMBOL(kernel_clone); # in /kernel/fork.c
# EXPORT_SYMBOL(do_execve); # in /fs/exec.c
```

```
# EXPORT_SYMBOL(getname_kernel); # in /fs/namei.c
# EXPORT_SYMBOL(do_wait); # in /kernel/exit.c

make # to build the kernel image and modules
make modules_install
make install
reboot # after reboot, the kernel has been updated
```

Now we can use the functions listed above. The development environment has been set up already.

2 Program 1

2.1 Implementing the child process

This part implements the child process. We get the file name to be executed by modifying the **argv** and we use the function **getpid()** to get the pid of the child process. Then we use **execve()** function to execute the test program. The details are shown below.

```
/* child process */
if (pid == 0)
{
    int i;
    char *arg[argc];

    for (i = 0; i < argc - 1; i++)
    {
        arg[i] = argv[i + 1];
    }
    arg[argc - 1] = NULL;
    /* get the test program path and name */

    printf("I'm the Child Process, my pid = %d\n", getpid());
    printf("Child process start to execute test program:\n");

    /* execute test program */
    execve(arg[0], arg, NULL);
}
```

2.2 Implementing the parent process

This part implements the parent process. After **fork()**, we should use function **waitpid()** to wait for the child process to terminate or stop. Here the option is

WUNTRACED, shows that we treat the stopped program as a terminated one, so that we can get **SIGSTOP** signal.

After waiting, we will get the status. To process the status, we get the signal from status by getting the last 7 bits, and we also use this variable to show whether this is a stopped program or not. If yes, we further get the stop signal. The details are shown below.

```
/* parent process */
else
{
    printf("I'm the Parent Process, my pid = %d\n", getpid());

    /* wait for child process terminates */
    waitpid(pid, &status, WUNTRACED);
    printf("Parent process receives SIGCHLD signal\n");

    /* check child process' termination status and signal */
    int signal = status & 0x7F; // get signal from lowest 7 bits
    int stop_spec = (((unsigned)status) >> 8);

    /* more codes here */
}
```

To output the signals, we use an array to store all the signals in consistent with their signal numbers. Firstly we judge whether it is normal terminated. If not, we further judge whether it is stopped or not. Further, we see it is terminated by other signals and we specifiy them using our array. The details are shown below.

```
/* parent process */
else
{
    /* previous codes */

    char signal_array[32][15] = {
        "normal", "SIGHUP", "SIGINT", "SIGQUIT", "SIGILL",
        "SIGTRAP", "SIGABRT", "SIGBUS", "SIGFPE", "SIGKILL",
        "SIGUSR1", "SIGSEGV", "SIGUSR2", "SIGPIPE", "SIGALRM",
        "SIGTERM", "unused", "SIGCHLD", "SIGCONT", "SIGSTOP",
        "SIGTSTP", "SIGTTIN", "SIGTTOU", "SIGURG", "SIGXCPU",
        "SIGXFSG", "SIGVTALRM", "SIGPROF", "SIGWINCH", "SIGIO",
        "SIGPWR", "SIGSYS"};

    switch (signal)
    {
    case 0: // normal termination
        printf("Normal termination with EXIT STATUS = 0\n");
        break;
```

```
20      case 127: // stop
21          printf("child process get stop signal\n");
22          break;
23      default:
24          printf("child process get %s signal\n",
                signal_array[signal]);
25          break;
26      }
27
28      if (signal == 127)
29      {
30          printf("child process stopped with signal = %d\n",
                stop_spec);
31      }
32      else if (signal != 0)
33      {
34          printf("child process terminated by %s signal with
                signal = %d\n", signal_array[signal], signal);
35      }
36  }
```

The whole codes are attached, including error detection and some other basic things.

2.3 Program output

This part shows 15 sample outputs with different signals, from Figure 1 to 15.

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 normal
Process start to fork
I'm the Parent Process, my pid = 4768
I'm the Child Process, my pid = 4769
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 1: normal signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 hangup
Process start to fork
I'm the Parent Process, my pid = 4634
I'm the Child Process, my pid = 4635
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
child process terminated by SIGHUP signal with signal = 1
```

Figure 2: hangup signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 interrupt
Process start to fork
I'm the Parent Process, my pid = 4710
I'm the Child Process, my pid = 4711
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
child process terminated by SIGINT signal with signal = 2
```

Figure 3: interrupt signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 quit
Process start to fork
I'm the Parent Process, my pid = 4823
I'm the Child Process, my pid = 4824
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
child process terminated by SIGQUIT signal with signal = 3
```

Figure 4: quit signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 illegal_instr
Process start to fork
I'm the Parent Process, my pid = 4666
I'm the Child Process, my pid = 4667
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
child process terminated by SIGILL signal with signal = 4
```

Figure 5: illegal_instr signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 trap
Process start to fork
I'm the Parent Process, my pid = 5016
I'm the Child Process, my pid = 5017
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
child process terminated by SIGTRAP signal with signal = 5
```

Figure 6: trap signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 abort
Process start to fork
I'm the Parent Process, my pid = 4471
I'm the Child Process, my pid = 4472
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
child process terminated by SIGABRT signal with signal = 6
```

Figure 7: abort signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 bus
Process start to fork
I'm the Parent Process, my pid = 4551
I'm the Child Process, my pid = 4552
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
child process terminated by SIGBUS signal with signal = 7
```

Figure 8: bus signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 floating
Process start to fork
I'm the Parent Process, my pid = 4592
I'm the Child Process, my pid = 4593
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
child process terminated by SIGFPE signal with signal = 8
```

Figure 9: floating signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 kill
Process start to fork
I'm the Parent Process, my pid = 4739
I'm the Child Process, my pid = 4740
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
child process terminated by SIGKILL signal with signal = 9
```

Figure 10: kill signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 segment_fault
Process start to fork
I'm the Parent Process, my pid = 4896
I'm the Child Process, my pid = 4897
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
child process terminated by SIGSEGV signal with signal = 11
```

Figure 11: segment_fault signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 pipe
Process start to fork
I'm the Parent Process, my pid = 2525
I'm the Child Process, my pid = 2526
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
child process terminated by SIGPIPE signal with signal = 13
```

Figure 12: pipe signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 alarm
Process start to fork
I'm the Parent Process, my pid = 4519
I'm the Child Process, my pid = 4520
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
child process terminated by SIGALRM signal with signal = 14
```

Figure 13: alarm signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 terminate
Process start to fork
I'm the Parent Process, my pid = 4972
I'm the Child Process, my pid = 4973
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
child process terminated by SIGTERM signal with signal = 15
```

Figure 14: terminate signal in Program 1

```
vagrant@csc3150:~/csc3150/Assignment_1_121090642/source/program1$ ./program1 stop
Process start to fork
I'm the Parent Process, my pid = 4928
I'm the Child Process, my pid = 4929
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receives SIGCHLD signal
child process get stop signal
child process stopped
```

Figure 15: stop signal in Program 1

3 Program 2

3.1 Prepare structs and functions

Before implementing, we should firstly prepare the functions and structs we need. The **`task`** struct is used to store the task we create. The **`wait_opts`** struct is for the **`do_wait()`**function parameter usage. The funtions with **`extern`** are the functions from Linux kernel. In additon, the functions used by myself are declared. The details are shown below.

```
1  /* for kthread_create */
2  static struct task_struct *task;
3
4  /* for do_wait parameters usage */
5  struct wait_opts
6  {
7      enum pid_type wo_type;
8      int wo_flags;
9      struct pid *wo_pid;
10
11     struct waitid_info *wo_info;
12     int wo_stat;
13     struct rusage *wo_rusage;
14
15     wait_queue_entry_t child_wait;
16 };
17
```

```
18 /* exported functions from kernel */
19 extern pid_t kernel_clone(struct kernel_clone_args *args);
20 extern int do_execve(struct filename *filename,
21                      const char __user *const __user *__argv,
22                      const char __user *const __user *__envp);
23 extern struct filename *getname_kernel(const char *filename);
24 extern long do_wait(struct wait_opts *wo);
25
26 /* func declaration */
27 int my_execve(void);
28 int my_fork(void *argc);
```

3.2 Implementing init function

This part is to initalize the module. In this part, we use **kthread_create()** to create a task and store the task in a variable. Then we wake up this task using the function **wake_up_process()**. The details are shown below.

```
1  static int __init program2_init(void)
2  {
3
4      printk("[program2] : module_init\n");
5
6      /* create a kernel thread to run my_fork */
7      printk("[program2] : module_init create kthread start\n");
8      task = kthread_create(&my_fork, NULL, "ZiyuThread");
9
10     /* wake up new thread if ok */
11     if (!IS_ERR(task))
12     {
13         printk("[program2] : module_init kthread start\n");
14         wake_up_process(task);
15     }
16
17     return 0;
18 }
```

3.3 Implementing my_fork function: kernel_clone

The last part creates a task and wake up the function **my_fork()**. In this function, we should fork a process using **do_fork()** if we use the previous version. However, now this function has been changed to **kernel_clone()**.

We firstly create a struct to store the parameters for this function. Here a series of flag bits can be set through the flags parameter. For example, **CLONE_VM**

indicates the shared process address space, **CLONE_UNTRACED** indicates that there is no need to track the child process, and **CSIGNAL** indicates the signal when the child process exits.

The **pidfd** is a flag used to indicate whether to create a pidfd file descriptor when cloning a new process. **child_pid** is used to point to user space memory in child process address space, similarly does **parent_pid**.

The **stack** should point to the location of the function to execute. The details are shown below.

```
1
2  // implement fork function
3  int my_fork(void *argc)
4  {
5      // set default sigaction for current process
6      /* previous codes here */
7
8      /* for kernel_clone parameter usage */
9      struct kernel_clone_args clone_args = {
10         .flags = ((lower_32_bits(SIGCHLD) | CLONE_VM |
               CLONE_UNTRACED) & ~CSIGNAL),
11         .pidfd = NULL,
12         .child_tid = NULL,
13         .parent_tid = NULL,
14         .exit_signal = (lower_32_bits(SIGCHLD) & CSIGNAL),
15         .stack = (unsigned long)&my_execve,
16         .stack_size = 0,                // normally set as 0
               because it is unused
17         .tls = 0
18     };
19
20     /* fork a process using kernel_clone or kernel_thread */
21     pid = kernel_clone(&clone_args);
22     printk("[program2] : The child process has pid = %d\n",
           pid);
23     printk("[program2] : This is the parent process, pid =
           %d\n", (int)current->pid);
24
25     /* more codes here */
26 }
```

3.4 Implementing my_execve function

This part will execute the file using **do_execve()**, which is easy to implement.

```
1  /* execute the test program */
2  int my_execve(void)
```

```
3  {
4      int exe_res;
5      const char path[] = "/tmp/test";
6
7      struct filename *file_name = getname_kernel(path);
8
9      /* execute a test program in child process */
10     exe_res = do_execve(file_name, NULL, NULL);
11
12     return 0;
13 }
```

3.5 Completing my_fork function: do_wait

After **kernel_clone()** shown in 3.3, this program is not completed, because we should wait for the signal using the function **do_wait()** and output it.

We set the **wait_opts** corretly with a variable status, which will be used to catch the status. The **PIDTYPE_PID** shows the type should be pid. The flags shows that whether the child process exits or stops, we will wait successfully.

After we get the status, it is easy to output the information since it is quite similar to program 1. The details are shown as below.

```
1  // implement fork function
2  int my_fork(void *argc)
3  {
4      /* previous codes here */
5
6      /* wait_opts paramters */
7      int status;
8
9      struct pid *wo_pid = NULL;
10     wo_pid = find_get_pid(pid);
11
12     /* wait_opts settings */
13     struct wait_opts do_wo = {
14         .wo_type = PIDTYPE_PID,
15         .wo_flags = WEXITED | WUNTRACED,
16         .wo_pid = wo_pid,
17         .wo_info = NULL,
18         .wo_stat = (int __user *)&status,
19         .wo_rusage = NULL,
20     };
21
22     /* wait until child process terminates */
23     printk("[program2] : child process\n");
24
```

```
25      int a;
26      a = do_wait(&do_wo);
27
28      put_pid(wo_pid);
29
30      status = do_wo.wo_stat; // get the status
31
32      /* check child process' termination status and signal */
33      int signal = status & 0x7F; // get signal from lowest 7 bits
34      int stop_spec = (((unsigned)status) >> 8);
35
36      /* more codes here similar to program1 */
37  }
```

3.6 Run program 2

To run program 2, follow the command lines below. Note that the test file should be preprared in advance in **/tmp/test**.

```
1  sudo su # get the root permission
2  make
3  insmod program2.ko
4  rmmod program2.ko
5  dmesg
```

3.7 Program output

This part shows 15 sample outputs with different signals, from Figure 16 to 30.

```
[ 2114.705173] [program2] : module_init
[ 2114.707191] [program2] : module_init create kthread start
[ 2114.710345] [program2] : module_init kthread start
[ 2114.712940] [program2] : The child process has pid = 14408
[ 2114.716266] [program2] : This is the parent process, pid = 14407
[ 2114.719538] [program2] : child process
[ 2114.722170] [program2] : Normal termination with EXIT STATUS = 0
[ 2114.726050] [program2] : The return status is 0
[ 2114.729021] [program2] : The return signal is 0
[ 2115.737076] [program2] : module_exit
```

Figure 16: normal signal in Program 2

```
[ 1808.085295] [program2] : module_init
[ 1808.087504] [program2] : module_init create kthread start
[ 1808.090673] [program2] : module_init kthread start
[ 1808.098773] [program2] : The child process has pid = 14198
[ 1808.101081] [program2] : This is the parent process, pid = 14196
[ 1808.103690] [program2] : child process
[ 1808.105279] [program2] : get SIGHUP signal
[ 1808.107264] [program2] : child process terminated
[ 1808.109498] [program2] : The return status is 1
[ 1808.111818] [program2] : The return signal is 1
[ 1809.416949] [program2] : module_exit
```

Figure 17: hangup signal in Program 2

```
[ 1895.391419] [program2] : module_init
[ 1895.393407] [program2] : module_init create kthread start
[ 1895.396861] [program2] : module_init kthread start
[ 1895.407096] [program2] : The child process has pid = 14297
[ 1895.410168] [program2] : This is the parent process, pid = 14296
[ 1895.413962] [program2] : child process
[ 1895.416500] [program2] : get SIGINT signal
[ 1895.419223] [program2] : child process terminated
[ 1895.421829] [program2] : The return status is 2
[ 1895.424442] [program2] : The return signal is 2
[ 1896.503309] [program2] : module_exit
```

Figure 18: interrupt signal in Program 2

```
[ 2327.399131] [program2] : module_init
[ 2327.401600] [program2] : module_init create kthread start
[ 2327.405548] [program2] : module_init kthread start
[ 2327.409571] [program2] : The child process has pid = 14506
[ 2327.413936] [program2] : This is the parent process, pid = 14505
[ 2327.417741] [program2] : child process
[ 2327.560363] [program2] : get SIGQUIT signal
[ 2327.561842] [program2] : child process terminated
[ 2327.563340] [program2] : The return status is 131
[ 2327.564858] [program2] : The return signal is 3
[ 2328.496612] [program2] : module_exit
```

Figure 19: quit signal in Program 2

```
[ 1860.667356] [program2] : module_init
[ 1860.669565] [program2] : module_init create kthread start
[ 1860.673082] [program2] : module_init kthread start
[ 1860.676912] [program2] : The child process has pid = 14241
[ 1860.679374] [program2] : This is the parent process, pid = 14240
[ 1860.681685] [program2] : child process
[ 1860.851443] [program2] : get SIGILL signal
[ 1860.852495] [program2] : child process terminated
[ 1860.853687] [program2] : The return status is 132
[ 1860.854904] [program2] : The return signal is 4
[ 1861.936057] [program2] : module_exit
```

Figure 20: illegal_instr signal in Program 2

```
[ 2573.647699] [program2] : module_init
[ 2573.650642] [program2] : module_init create kthread start
[ 2573.654655] [program2] : module_init kthread start
[ 2573.658270] [program2] : The child process has pid = 14771
[ 2573.662443] [program2] : This is the parent process, pid = 14770
[ 2573.665685] [program2] : child process
[ 2573.868371] [program2] : get SIGTRAP signal
[ 2573.869998] [program2] : child process terminated
[ 2573.871813] [program2] : The return status is 133
[ 2573.873648] [program2] : The return signal is 5
[ 2574.737775] [program2] : module_exit
```

Figure 21: trap signal in Program 2

```
[ 1587.784880] [program2] : module_init
[ 1587.787018] [program2] : module_init create kthread start
[ 1587.795358] [program2] : module_init kthread start
[ 1587.798779] [program2] : The child process has pid = 14014
[ 1587.802841] [program2] : This is the parent process, pid = 14013
[ 1587.806696] [program2] : child process
[ 1587.954702] [program2] : get SIGABRT signal
[ 1587.956331] [program2] : child process terminated
[ 1587.958036] [program2] : The return status is 134
[ 1587.959816] [program2] : The return signal is 6
[ 1589.413876] [program2] : module_exit
```

Figure 22: abort signal in Program 2

```
[ 1710.282369] [program2] : module_init
[ 1710.284821] [program2] : module_init create kthread start
[ 1710.289393] [program2] : module_init kthread start
[ 1710.292268] [program2] : The child process has pid = 14050
[ 1710.294605] [program2] : This is the parent process, pid = 14049
[ 1710.297398] [program2] : child process
[ 1710.430430] [program2] : get SIGBUS signal
[ 1710.431514] [program2] : child process terminated
[ 1710.432680] [program2] : The return status is 135
[ 1710.433839] [program2] : The return signal is 7
[ 1711.587759] [program2] : module_exit
```

Figure 23: bus signal in Program 2

```
[ 1752.100063] [program2] : module_init
[ 1752.102831] [program2] : module_init create kthread start
[ 1752.117295] [program2] : module_init kthread start
[ 1752.119583] [program2] : The child process has pid = 14104
[ 1752.120959] [program2] : This is the parent process, pid = 14103
[ 1752.122595] [program2] : child process
[ 1752.257524] [program2] : get SIGFPE signal
[ 1752.258676] [program2] : child process terminated
[ 1752.259912] [program2] : The return status is 136
[ 1752.261096] [program2] : The return signal is 8
[ 1752.999250] [program2] : module_exit
```

Figure 24: floating signal in Program 2

```
[ 1937.802829] [program2] : module_init
[ 1937.804786] [program2] : module_init create kthread start
[ 1937.808146] [program2] : module_init kthread start
[ 1937.810415] [program2] : The child process has pid = 14352
[ 1937.813527] [program2] : This is the parent process, pid = 14351
[ 1937.815798] [program2] : child process
[ 1937.818558] [program2] : get SIGKILL signal
[ 1937.821007] [program2] : child process terminated
[ 1937.823769] [program2] : The return status is 9
[ 1937.826666] [program2] : The return signal is 9
[ 1938.840830] [program2] : module_exit
```

Figure 25: kill signal in Program 2

```
[ 2394.222825] [program2] : module_init
[ 2394.224442] [program2] : module_init create kthread start
[ 2394.227082] [program2] : module_init kthread start
[ 2394.230496] [program2] : The child process has pid = 14586
[ 2394.233486] [program2] : This is the parent process, pid = 14584
[ 2394.237603] [program2] : child process
[ 2394.373475] [program2] : get SIGSEGV signal
[ 2394.375051] [program2] : child process terminated
[ 2394.376721] [program2] : The return status is 139
[ 2394.378459] [program2] : The return signal is 11
[ 2396.109570] [program2] : module_exit
```

Figure 26: segment_fault signal in Program 2

```
[ 2288.788750] [program2] : module_init
[ 2288.790391] [program2] : module_init create kthread start
[ 2288.792213] [program2] : module_init kthread start
[ 2288.794000] [program2] : The child process has pid = 14466
[ 2288.800997] [program2] : This is the parent process, pid = 14464
[ 2288.803448] [program2] : child process
[ 2288.804849] [program2] : get SIGPIPE signal
[ 2288.806343] [program2] : child process terminated
[ 2288.807949] [program2] : The return status is 13
[ 2288.809428] [program2] : The return signal is 13
[ 2290.143083] [program2] : module_exit
```

Figure 27: pipe signal in Program 2

```
[ 1510.961847] [program2] : module_init
[ 1510.963363] [program2] : module_init create kthread start
[ 1510.966524] [program2] : module_init kthread start
[ 1510.969062] [program2] : The child process has pid = 13914
[ 1510.972470] [program2] : This is the parent process, pid = 13913
[ 1510.975827] [program2] : child process
[ 1512.974671] [program2] : get SIGALRM signal
[ 1512.985796] [program2] : child process terminated
[ 1512.994076] [program2] : The return status is 14
[ 1513.002143] [program2] : The return signal is 14
[ 1518.905642] [program2] : module_exit
```

Figure 28: alarm signal in Program 2

```
[ 2507.644886] [program2] : module_init
[ 2507.647073] [program2] : module_init create kthread start
[ 2507.650768] [program2] : module_init kthread start
[ 2507.653727] [program2] : The child process has pid = 14696
[ 2507.657617] [program2] : This is the parent process, pid = 14695
[ 2507.661253] [program2] : child process
[ 2507.663607] [program2] : get SIGTERM signal
[ 2507.665902] [program2] : child process terminated
[ 2507.668303] [program2] : The return status is 15
[ 2507.671025] [program2] : The return signal is 15
[ 2509.740559] [program2] : module_exit
```

Figure 29: terminate signal in Program 2

```
[ 1490.577120] [program2] : module_init
[ 1490.579293] [program2] : module_init create kthread start
[ 1490.588291] [program2] : module_init kthread start
[ 1490.590417] [program2] : The child process has pid = 10092
[ 1490.593450] [program2] : This is the parent process, pid = 10091
[ 1490.596678] [program2] : child process
[ 1490.598726] [program2] : get stop signal
[ 1490.600928] [program2] : child process stopped
[ 1490.603426] [program2] : The return signal is 19
[ 1490.605999] [program2] : The return status is 4991
[ 1491.832908] [program2] : module_exit
```

Figure 30: stop signal in Program 2

4 Learn from Tasks

From the two tasks, I have learnt:

1. How to check the version of the Linux distributions and Linux kernel version (By using `lsb_release -a` and `uname -r`). In additon, I also understand some common Linux commands.

2. How to change or update the kernel version. And in addition, how to modify the kernel files and recompile in adapt to our works.

3. How to create a process both in user mode and kernel mode. Learn some basic steps and the corresponding functions, with the parameters to set.

5. How different status should be treated to catch the signals. (Here we directly learn them from the kernel function library.)

4. How to insert and remove kernel object modules. How the modules being initalized and exited.