Task 1

Program design

In this task, the program (***program1.c***) is going to fork a child process to execute the test program, then wait for the return status of the child process and finally report the information of the process.

To do the task, first, fork a child process:

```
printf("Process start to fork\n");
pid_t pid;
pid = fork();
```

Then, the program could be divided into two parts for parent process and child process respectively:

- For child process:
  - Announce the pid

    ```
    printf("I'm the child process, my pid = %d\n", getpid());
    ```

  - Read the arguments to get the test file and its input arguments

    ```
    int i;
    char *arg[argc];
    // read the arguments of the test file
    for(i = 0; i < argc; i++) {
      arg[i] = argv[i+1];
    }
    arg[argc-1] = NULL;
    ```

  - Execute the test file

    ```
    execve(arg[0], arg, NULL); // arg[0] is the test file
    ```

- For parent process:
  - Annouce the pid

    ```
    printf("I'm the parent process, my pid = %d\n", getpid());
    ```

  - Wait for the return status of the child process and announce if successfully receives

```
waitpid(pid, &status, WUNTRACED);
printf("Parent process receiving the SIGCHLD signal\n");
```

- According to the status, announce the termination type and the signal raised in the child process

  (this program is able to handle the 15 signals that provided by the test programs)

  ```
  if (WIFEXITED(status)) {
    printf("Normal termination with EXIT STATUS = %d\n",
  WEXITSTATUS(status));
  }else if(WIFSIGNALED(status)) {
    switch(WTERMSIG(status)){
      case 1:
        printf("child process get SIGHUP signal\nchild process is hang
  up by hang up          signal\n"); break;
      case 2:
        ...
      ...
  ...
  ```

The above is the basic logic of the program1. More details of the code please refer to ***program1.c***.

Running environment

- OS: *Ubuntu-16.04-32bit*
- Kernel: *Linux-4.10.14*

Steps to run the program

1. In the directory of program1, type the following command and enter.

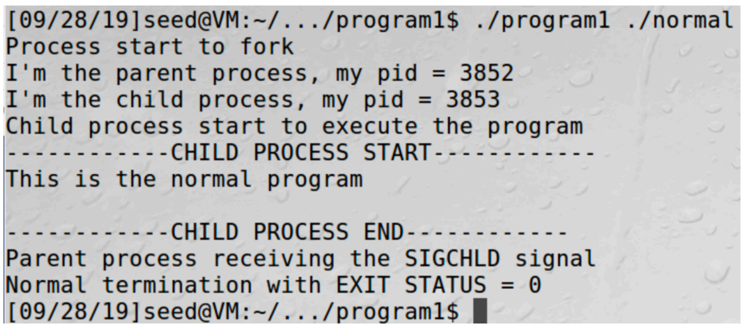
   ```
   $make
   ```

2. In the directory of program1, type the following command and enter,

   ```
   $./program1 TEST_FILE_NAME ARG1 ARG2 ...
   ```

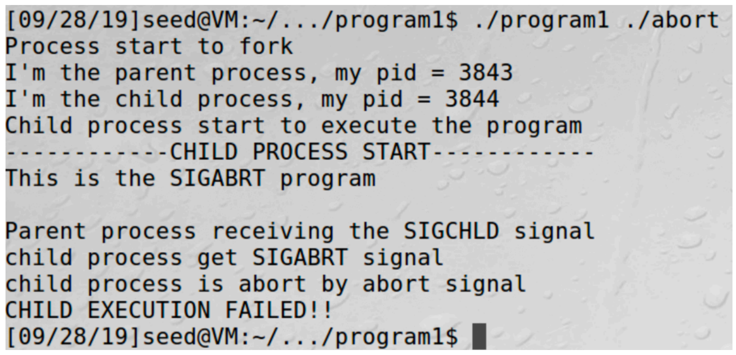
   where TEST_FILE_NAME is the name of the test program and ARG1, ARG2 ... are the arguments that the test program could have.

Program output



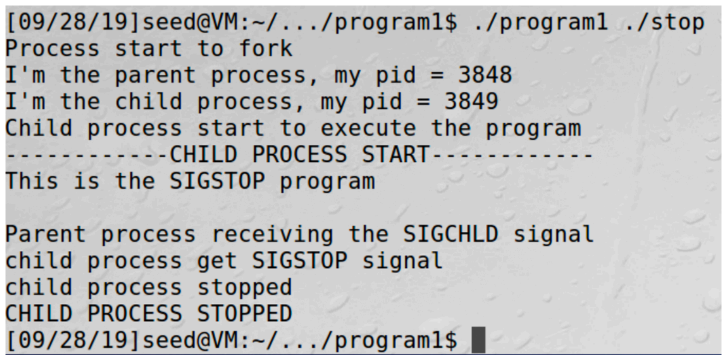
```
[09/28/19]seed@VM:~/.../program1$ ./program1 ./normal
Process start to fork
I'm the parent process, my pid = 3852
I'm the child process, my pid = 3853
Child process start to execute the program
------------CHILD PROCESS START------------
This is the normal program

------------CHILD PROCESS END------------
Parent process receiving the SIGCHLD signal
Normal termination with EXIT STATUS = 0
[09/28/19]seed@VM:~/.../program1$ █
```



```
[09/28/19]seed@VM:~/.../program1$ ./program1 ./abort
Process start to fork
I'm the parent process, my pid = 3843
I'm the child process, my pid = 3844
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGABRT program

Parent process receiving the SIGCHLD signal
child process get SIGABRT signal
child process is abort by abort signal
CHILD EXECUTION FAILED!!
[09/28/19]seed@VM:~/.../program1$ █
```



```
[09/28/19]seed@VM:~/.../program1$ ./program1 ./stop
Process start to fork
I'm the parent process, my pid = 3848
I'm the child process, my pid = 3849
Child process start to execute the program
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receiving the SIGCHLD signal
child process get SIGSTOP signal
child process stopped
CHILD PROCESS STOPPED
[09/28/19]seed@VM:~/.../program1$ █
```

The above screen shots are the running results with three test programs: *normal.c, abort.c, stop.c*.

What I learned

How to fork a process under the user mode

- Use *pid* to distinguish the parent process and the child process.
- Use *printf()* to trace the process
- Use *execve()* to execute the input file
- Use wait to grab the termination status of the child process
- Understand the running logic between process
- ...

Task 2

Program design

In this task, we designed a program (***program2.c***) as kernel module. If we initialize it, it is going to create a kernel thread. In this created thread, it would fork a child process to execute the test program and wait for the return status of the child process and finally report the information of the process.

Therefore, to do the task:

1. In the *program2_init* function, create a thread to run function *my_fork*:

   ```
   struct task_struct *task;
   task = kthread_create(&my_fork, NULL, "MyThread");
   ```

2. After the thread has been created, in the *my_fork* function:

   1. Make a system call *_do_fork* , fork a child process and let it execute the function *my_exec* :

      ```
      pid_t pid;
      pid = _do_fork(SIGCHLD, (unsigned long)&my_exec, 0, NULL, NULL, 0);
      ```

      - In the function *my_exec()*, set the path, arguments and environment of the test program. Then get the test program name by the path and execute it by system call *do_execve*:

        ```
        int result;
        const char path[] = "/home/seed/Documents/program2/test";
        const char *const argv[] = {path, NULL, NULL};
        const char *const envp[] = {"Home=/",
        "PATH=/sbin:/user/sbin:/bin:/user/bin",
        NULL};
        struct filename * my_filename = getname(path);
        printk("[program2] : child process\n");
        result = do_execve(my_filename, argv, envp);
        ```

        Here, the path of the test program is an absolute path, which means, to successfully run the test program, user should keep the test file under the correct path or reset the path properly .

   2. Annouce the pid of the parent process and the 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);
      ```

   3. To wait for the return status of the child process, call function *my_wait*

      ```
      my_wait(pid);
      ```

- In the *my_wait(pid)* function:
  1. Create a *wait_opts* structure:

     ```
     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 = WEXITED;
     wo.wo_info = NULL;
     wo.wo_stat = (int __user*)&status;
     wo.wo_rusage = NULL;
     ```

  2. Make a system call *do_wait* , it will wait for the return status of the child process:

     ```
     int a;
     a = do_wait(&wo);
     ```

  3. According to the return status recorded in *wo.wo_stat*, we could identify the SIGNAL raised in the child process and then we could report it:

     ```
     if (*wo.wo_stat == 7){
         printk("[program2] : get SIGBUS signal\n");
         printk("[program2] : child process has bus error\n");
         printk("[program2] : The return signal is 7\n");
     }
     ...
     ```

  4. Decrease the count and free memory:

     ```
     put_pid(wo_pid);
     ```

- For struct *wait_opts* and *do_wait*, to make the program compile smoothly, add the definition and declaration of them at the beginning, which are copied from the source code in kernel:

  ```
  struct wait_opts {
      enum pid_type         wo_type;
      int               wo_flags;
      struct pid         *wo_pid;

      struct siginfo __user     *wo_info;
  ```

```
    int __user          *wo_stat;
    struct rusage __user     *wo_rusage;

    wait_queue_t         child_wait;
    int             notask_error;
};

extern long do_wait(struct wait_opts *wo);
```

Running environment *

- OS: *Ubuntu-16.04-32bit*
- Kernel: *Linux-4.10.14* (with modification)
    - There are some non-static functions used in this program (*_do_fork, do_execve, getname, do_wait*), we should firstly export these symbols so that they can be used in the kernel module.
        1. Under the directory of the kernel file, gain the root access:

           ```
           $sudo su
           ```

        2. Go to the source code file of the target symbol:

           ```
           $cd PATH_OF_FILE
           $gedit FILE_NAME.c
           ```

           where the PATH_OF_FILE/FILE_NAME.c for each symbol is:

           - *_do_fork* : /kernel/fork.c
           - *do_execve* : /fs/exec.c
           - *getname* : /fs/namei.c
           - *do_wait* : /kernel/exit.c
        3. The source code file would be opened after #2. Then in the source code file, under the implementation of each target functions, add one line of the following code:

           ```
           EXPORT_SYMBOL_GPL(SYMBOL_NAME)
           ```

           where SYMBOL_NAME is the name of the function: *_do_fork, do_execve, getname, do_wait*.

        4. Back to the directory of the kernel file, rebuild the kernel by execute the following commands one by one:

```
$make bzImage
$make modules
$make modules_install
$make install
$reboot
```

5. Before using these symbols in our kernel module, use 'extern' to clarify them. Then we could use the symbols we exported.

Steps to run the program

1. Compile the test file:

   ```
   gcc FILE_NAME.c -o FILE_NAME
   ```

   where the FILE_NAME is the file name of the test file

2. In the directory of program2, type the following command and enter to compile files:

   ```
   $make
   ```

3. In the directory of program2, type the following command and enter to insert the kernel module:

   ```
   $insmod program2.ko
   ```

4. In the directory of program2, type the following command and enter to remove the kernel module:

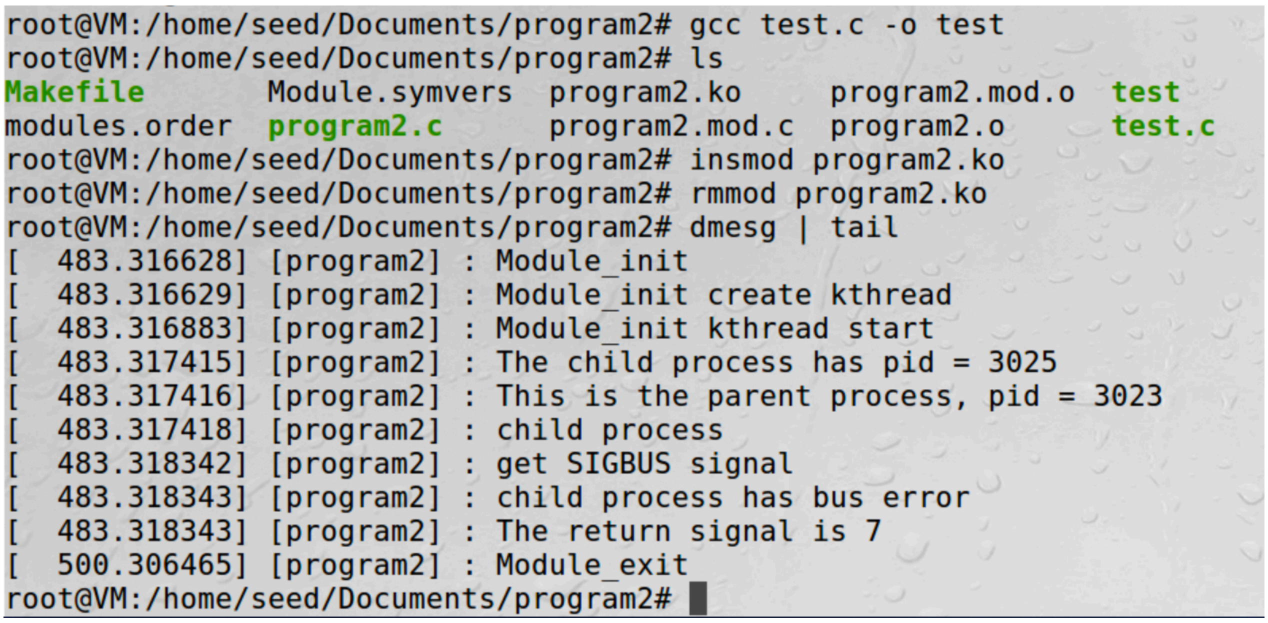
   ```
   $rmmod program2.ko
   ```

5. In the directory of program2, type the following command and enter to see the latest messages appear:

   ```
   $dmesg | tail
   ```

Program output

The following screen shot is the running results with the test file provided.



```
root@VM:/home/seed/Documents/program2# gcc test.c -o test
root@VM:/home/seed/Documents/program2# ls
Makefile       Module.symvers  program2.ko     program2.mod.o  test
modules.order  program2.c      program2.mod.c  program2.o      test.c
root@VM:/home/seed/Documents/program2# insmod program2.ko
root@VM:/home/seed/Documents/program2# rmmod program2.ko
root@VM:/home/seed/Documents/program2# dmesg | tail
[  483.316628] [program2] : Module_init
[  483.316629] [program2] : Module_init create kthread
[  483.316883] [program2] : Module_init kthread start
[  483.317415] [program2] : The child process has pid = 3025
[  483.317416] [program2] : This is the parent process, pid = 3023
[  483.317418] [program2] : child process
[  483.318342] [program2] : get SIGBUS signal
[  483.318343] [program2] : child process has bus error
[  483.318343] [program2] : The return signal is 7
[  500.306465] [program2] : Module_exit
root@VM:/home/seed/Documents/program2# 
```

What I learned

- How to compile a kernel
- How to insert and remove a kernel module into kernel
- How to read and modify the source code of kernel
- How to create a kernel module and what can be done in the module
    - how to create *kthread*
    - how to fork a process under kernel mode (*_do_fork*)
    - how to execute a file under kernel mode (*do_execve*)
    - how to grab the termination status of the child process under kernel mode (*do_wait*)
- More understanding the philosophy of the kernel
- ...