CSC3150 Assignment1

119010054 Dai Yulong

1. Design Ideas

1.1 Overview

The assignment1 mainly focuses on multi-processes programming. It's required to master following key points:

- How to create processes and threads: The program executes parent and child processes by using `pid` to distinguish them.
- How user mode and the kernel interact with each other.
- How processes communicate with each other: By using shared memory (which is implemented by `mmap()` and `munmap()`, a segment of memory is allocated and all the processes can read and write towards this memory.

1.2 Design Ideas of Task1

In the program1, the first step is to create a process using function `fork()`. It is called once but returns twice. The only difference between the two return value is that the child process returns 0 and the parent process returns the child process ID. In this case, `if` loop is used to judge whether the running process is a father process or a child process. Three situations exit. `pid = -1` means that something wrong occurs when creating child process. `pid = 0` means that the running process is the child process. In this process, firstly the program prints out the current pid and then execute the input file using function `execve()`.

For the second step, in the parent process, function `waitpid()` is used to wait the end of the child process. The reference to this function is `waitpid(pid, &status, WUNTRACED)`. The option is `WUNTRACED`, which reports the return signals of child process (also including the case of stop). If the option is not `WUNTRACED`, the waitpid function will keep on waiting the status of the child process when the child process raises the `SIGSTOP` signal.

After getting the terminated status, the program print it out. The program first uses `WIFEXITED(status)` to judge whether the child process terminates normally (return `True` when child process normally terminated). Then the program uses `WIFSIGNALED` to judge whether it's `SIGSTOP` signal or not (renturn `True` when it's not `SIGSTOP` signal). Then `WTERMSIG(status)` is

used to distinguish different abnormal terminated signals. The correspondence table of status codes and terminated signal is attached below. Finally, `WIFSTOPPED(status)` is used to identify the `SIGSTOP` signal (return `True` when it's `SIGSTOP` signal).

1.3 Design Ideas of Task2

In the program2, the first thing to do is changing the kernel to export `_do_fork(), do_execve(), getname(), do_wait()`. It's required to go to the corresponding file of these four functions and check whether they're static. If the function is non-static, for example `_do_fork`, it's necessary to add `EXPORT_SYMBOL(_do_fork)`. After remaking the kernel, the program can use the above functions to under the kernel module. In the program, to use these functions, we need to declare them by `extern`. Notice that when using `do_wait()`, `wait_opts` struct needed to be declared in front.

After declaring these functions and structs, it's time to use them to create new threads and processes. The first step is to create a new thread in function `program2_init()` to initialize the kernel module. Since `&my_fork` is passed into the kthread_create function, the new thread will start from the address of the function `my_fork()`. Then, `wake_up_process()` is used to wake up the thread, which means the function `my_fork()` starts to be executed. Then the second step is to create child process in `my_fork()`. When creating new process, since I pass `&my_exec` into `_do_fork()`, the child process starts from the address of the function `my_fork()`. Then `my_exec` function is declared to execute the test files under specified path. After getting the filename, `do_execve()` function is used to run the test file. The test file raises certain signal which will be captured by father process.

Then the father process is executed in `my_wait()` which is called in `my_fork()`. The struct `wait_opts wo` is declared to get signals from the child process. The `wo_flags` of `wo` is assigned to `WEXITED | WUNTRACED`, which means the `do_wait()` function can get both `SIGSTOP` signal and other EXIT signals. When the child process is terminated or raise a signal, the father process stops waiting and judges what signal is raised by the child process. Then the information of the signals will be printed out in kernel log. Finally, the `wo_pid` is released through `put_pid()` function.

1.4 Design Ideas of Bonus

The bonus program can be divided into two part, which are creating child process and printing out a process tree. For the first part, my program support creating child process incrementally by using recursion. When typing `./myfork test_program1 test_program2 test_program3`, test_program_1 is parent of test_program_2, and test_program_2 is the parent of test_program_3 (it also support multiple executable files as the argument of `myfork`). The proceeding one is parent, and the tailing one is child. The function `void create_process(int argc, int num, char *arg[], int *pidId[], int *pidStatus[])` recurse itself to create child process incrementally. In this function, firstly, it `fork()` a child process. The child process uses `execve(arg[num],arg,NULL)` to execute the test_program. The parent process uses `waitpid()` to wait until the child process, then records each termination signal in `pidStatus[]`. In the child process, if `argc-1 > num + 1`, the function call itself to do the whole process again. In this way, a process tree can be achieved `Process tree: 61205->61206->61207->61208->61209->61210`, where the proceeding one is parent, and the tailing one is child.

For the second part, because process are independent, data cannot be shared, it's a problem to record all process information in one place. This program uses memory sharing mapping `mmap()` and `munmap()` to solve this problem. The basic idea is to apply for a piece of memory where every process can writes to or read the shared memory. The shared memory mapping area is created by `(int*)mmap(NULL, 40, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0)`, which adopts anonymous mapping (no need for extra auxiliary file). And it returns the first address of created shared memory. And the `munmap()` is used to release the memory mapping area in case of memory leak. After using `mmap()` and `munmap()` to store all information of processes in `pidId[]` and `pidStatus[]`, the program then print them out in kernel log.

2. Environments and Run Methods

2.1 Environments

My project adopts the recommended environment:

- Linux Version

```
dai@ubuntu:~$ cat /etc/issue
Ubuntu 16.04.5 LTS \n \l
```

- Linux Kernel Version

```
dai@ubuntu:~$ uname -r
4.10.14
```

- GCC Version

```
dai@ubuntu:~$ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

2.2 Run Methods

TASK1

- How to compile:
  Using `cd` to enter the `program1` directory, then type `make` command and enter.

  ```
  root@ubuntu:/home/dai/Desktop/program1# make
  ```

- How to clear:
  In the `program1` directory, type `make clean` command and enter.

  ```
  root@ubuntu:/home/dai/Desktop/program1# make clean
  ```

- How to execute:
  In the `program1` directory, type `./program1  TEST_CASE  ARG1  ARG2 ...`, where `TEST_CASE` is the name of test program and `ARG1, ARG2,...` are names of arguments that the test program could have.

  ```
  root@ubuntu:/home/dai/Desktop/program1# ./program1 ./normal
  ```

TASK2

The compile and clear methods are the same as **PROBLEM1**.

- How to execute:
  1.Type `sudo insmod program2.ko` under `program2` directory and enter
  2.Type `sudo rmmod program2` and enter to remove the program2 module.
  3.Type `dmesg | tail -n 10` and enter to display message buffer in kernel.
  (Notice: The current variable `path` in `my_exec()` is `/opt/test`. It's can be used to testing different cases.)

  ```
  root@ubuntu:/home/dai/Desktop/program2# insmod program2.ko
  root@ubuntu:/home/dai/Desktop/program2# rmmod program2.ko
  root@ubuntu:/home/dai/Desktop/program2# dmesg | tail -n 10
  ```

  (Particular attention：When testing case of `alarm`, we need to wait two seconds between insert and remove module operations. Because there exits `alarm(2)`. The alarm process will last two seconds and only after two seconds can the parent process recevice alarm signal.)

BONUS

The compile and clear methods are the same as **PROBLEM1**.

- How to execute:
  In the `bonus` directory, type `./myfork TEST_PRO1  TEST_PRO2  TEST_PRO3 ...`, where `TEST_PRO1, TEST_PRO2,...` are names of test programs.

  ```
  root@ubuntu:/home/dai/Desktop/bonus# ./myfork hangup normal1 kill normal2
  quit normal3
  ```

3. Sample Output

3.1 Sample Outputs of Task1

For task1, all fifteen test cases can be executed successfully. Below are three typical sample outputs:

- Output for normal termination:

```
root@ubuntu:/home/dai/Desktop/program1# ./program1 ./normal
Process start to fork
I'm the Parent Process, my pid = 56272
I'm the Child Process, my pid = 56273
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the normal program

------------CHILD PROCESS END------------
Parent process receiving the SIGCHILD signal!
Normal termination with EXIT STATUS = 0
```

- Output for stopped process:

```
root@ubuntu:/home/dai/Desktop/program1# ./program1 ./stop
Process start to fork
I'm the Parent Process, my pid = 56851
I'm the Child Process, my pid = 56852
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receiving the SIGCHILD signal!
child process get SIGSTOP signal
child process stopped
CHILD PROCESS STOPPED
```

- Output for killed process:

```
root@ubuntu:/home/dai/Desktop/program1# ./program1 ./kill
Process start to fork
I'm the Parent Process, my pid = 56864
I'm the Child Process, my pid = 56865
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGKILL program

Parent process receiving the SIGCHILD signal!
child process get SIGKILL signal
child process is killed
CHILD EXECUTION FAILED
```

3.2 Sample Outputs of Task2

For task2, all fifteen test cases in task1 can be executed successfully. Below are three typical sample outputs:

- Output for process which has bus error (test case):

```
root@ubuntu:/home/dai/Desktop/program2# dmesg | tail -n 10
[142565.295617] [program2] : module_init
[142565.298172] [program2] : module_init create kthread start
[142565.298270] [program2] : module_init kthread start
[142565.298539] [program2] : The child process has pid = 60200
[142565.298540] [program2] : This is the parent process, pid = 60198
[142565.298614] [program2] : child process
[142565.489390] [program2] : get SIGBUS signal
[142565.489391] [program2] : child process has bus error
[142565.489391] [program2] : The return signal is 7
[142568.677799] [program2] : module_exit./my
```

- Output for stopped process:

```
root@ubuntu:/home/dai/Desktop/program2# dmesg | tail -n 10
[142385.926255] [program2] : module_init
[142385.926297] [program2] : module_init create kthread start
[142385.926398] [program2] : module_init kthread start
[142385.926408] [program2] : The child process has pid = 59516
[142385.926409] [program2] : This is the parent process, pid = 59515
[142385.926417] [program2] : child process
[142385.929679] [program2] : get SIGSTOP signal
[142385.929680] [program2] : child process stopped
[142385.929681] [program2] : The return signal is 19
[142391.729315] [program2] : module_exit./my
```

- Output for terminated process:

```
root@ubuntu:/home/dai/Desktop/program2# dmesg | tail -n 10
[142745.805414] [program2] : module_init
[142745.806008] [program2] : module_init create kthread start
[142745.806506] [program2] : module_init kthread start
[142745.806524] [program2] : The child process has pid = 60885
[142745.806525] [program2] : This is the parent process, pid = 60883
[142745.806528] [program2] : child process
[142745.817535] [program2] : get SIGTERM signal
[142745.817536] [program2] : child process terminated
[142745.817537] [program2] : The return signal is 15
[142753.205806] [program2] : module_exit./my
```

3.3 Sample Output of Bonus

The bonus program supports multiple executable files as the input:

```
root@ubuntu:/home/dai/Desktop/bonus# ./myfork hangup normal1 kill normal2
terminate
------------CHILD PROCESS START------------
This is the SIGTERM program

This is normal2 program
```

```
------------CHILD PROCESS START------------
This is the SIGKILL program

This is normal1 program
------------CHILD PROCESS START------------
This is the SIGHUP program



---
Process tree: 61205->61206->61207->61208->61209->61210
Child process 61210 of parent process 61209 is terminated by signal 15
(Terminate)
Child process 61209 of parent process 61208 terminated normally with exit code 0
Child process 61208 of parent process 61207 is terminated by signal 9 (Kill)
Child process 61207 of parent process 61206 terminated normally with exit code 0
Child process 61206 of parent process 61205 is terminated by signal 1 (Hangup)
Myfork process (61205) terminated normally
```

3.x Correspondence Table of Status Codes and Signals

In this project, there're totally fifteen signals, which are normal signal, SIGSTOP, SIGABRT, SIGALRM, SIGBUS, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGTRAP. For the second signal SIGSTOP, it can be distinguished by **WIFSTOPPED(status)** (return 1). For the rest thirteen abnormal terminated signals, **WTERMSIG(status)** will return an exact number to distinguish them. The Correspondence table of status codes and signals is shown below:

Table 3.1 Correspondence Table of Status Codes and Signals

SIGNALS	STATUS CODES	SIGNALS	STATUS CODES
SIGABRT	6	SIGALRM	14
SIGBUS	7	SIGFPE	8
SIGHUP	1	SIGILL	4
SIGINT	2	SIGKILL	9
SIGPIPE	13	SIGQUIT	3
SIGSEGV	11	SIGTERM	15
SIGTRAP	5		
SIGSTOP	19		

4. Reflection

After finishing the assignment1, I learned several things about multi-processes programming, some basic knowledge about kernel, and interaction between user mode and kernel:

Comprehensive understanding many new functions:

- Several systems calls related to multi-processing programming.
- The usage of `_do_fork() , do_execve() , do_wait()` .... Besides, I also learn how these system calls are implemented in the kernel.
- Distinguish different signals and corresponding status codes. Also, I learned the difference between `wait()` and `waitpid()` and how the parent process uses them to receive and deal with the return status signals.
- Under kernel mode, learned about how to make a kernel object, how to insert it and remove it.

Communication between different processes

In the bonus part, I learn how to achieve the communications between different processes. There're actually two ways. And I choose Shared-memory method by using `mmap()` and `munmap()`.

- Shared-memory: In fact, the memory of different processes are distributed, which means they cannot communicate with each other. By

  using shared-memory method, I can maintain an array to record the information of all processes.

Special discovery when debugging

In the program2, when I test the case of `alarm`, I found that if I insert the module and remove the module too quick, the kernel log will report an error. After lots of tests, I found that the minimum interval between the two instructions was two-seconds. Then I supposed that it was due to the function `alarm(2)` in the test case. During that two-seconds, the parent process didn't receive the `alarm` signal. Actually the module init process didn't accomplish. And if we remove the module during that two-seconds, an error will definitely occur. During the process of solving this problem, I have a better understanding of creating module under kernel mode.