The Chinese University of Hong Kong, Shenzhen
CSC3150, 2022 Fall

Report of Assignment 1

Qiu Weilun 120090771

October 9, 2022

Date Performed ..... October 7, 2022

1 Task 1

The main task of task 1 is to create a child process under the current parent process. In the child process, the test program is executed. The child process may terminate normally, terminated by signals or stopped by signals. The raised signal would be sent back to the parent process and the corresponding information of the signals would be printed in the terminal. All the above task would be accomplished in user mode instead of kernel mode.

1.1 Design of the Program

To finish task 1, there would be two problems to be considered. Firstly, how the child process is created and how to seperate it from the parent process so that the behavior of the two process could be controlled seperately in user mode. In addition, how to implement the delivery of process signal information between the two processes.

The function `fork()` provide a useful way to solve the first problem. It create a new process under current process. The two processes are executed simultaneously after the API `fork()` is invoked. It will return the process ID (PID) of the child process. Since there are two processes working in the program, there would be two return values for the `fork()` API. One is the PID of the child process of current process. The other one is 0, which indicates that there is no child process of the currnet child process. Making use of this information, the child process can be seperated from the parent process by the defferent return values of `fork()`.

Since signals will be sent from the child process to the parent process, the parent process is going to wait for the test program to raise signals and terminate the child process. The `waitpid()` function may ask the parent process to wait. The prototype of function `waitpid()` is shown as following.

```
pid_t waitpid(pid_t pid, int *status, int options)
```

In this function, the parameter `int *status` will record the exit status of the child process. However, to get the actual signal codes of the corresponding signals, several functions are used.

```
int WIFEXITED(int status)   // check whether the process has exited
int WIFSIGNALED(int status) // the process is terminated by signals
int WIFSTOPPED(int status)  // the process is stopped by signals

int WEIXTSTATUS(int status) // check the normal exit status
int WTERMSIG(int status) // check the signal that terminate the process
int WSTOPSIG(int status) // check the signal that stop the signal
```

Using the above functions, all the signals from the child process can be handled properly. Since all the operations of program1 are conducted in user mode, it is not necessary to concern about kernel compilation.

1.2 Program Outputs

The following screenshots are the results of some of the test programs.

- result of test case abort.c

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./abort
Process start to fork.
I'm the Parent Process, my pid = 1965
I'm the Child Process, my pid = 1966
Child process start to execute test program.
------------CHILD PROCESS START------------
This is the SIGABRT program

Parent process receives SIGCHLD signal.
Child process raises SIGABRT signal.
Error is found in the program and the child process is terminated by abort().
```

- result of test case alarm.c

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./alarm
Process start to fork.
I'm the Parent Process, my pid = 2555
I'm the Child Process, my pid = 2556
Child process start to execute test program.
------------CHILD PROCESS START------------
This is the SIGALRM program

Parent process receives SIGCHLD signal.
Child process raises SIGALRM signal.
The child process is terminated by time signal.
```

- result of test case bus.c

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./bus
Process start to fork.
I'm the Parent Process, my pid = 2621
I'm the Child Process, my pid = 2622
Child process start to execute test program.
------------CHILD PROCESS START------------
This is the SIGBUS program

Parent process receives SIGCHLD signal.
Child process raises SIGBUS signal.
The child process is terminated out of bus error (or bad memory access).
```

- result of test case floating.c

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./floating
Process start to fork.
I'm the Parent Process, my pid = 2689
I'm the Child Process, my pid = 2690
Child process start to execute test program.
------------CHILD PROCESS START------------
This is the SIGFPE program

Parent process receives SIGCHLD signal.
Child process raises SIGFPE signal.
termination caused by floating-point exception
```

- result of test case hangup.c

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./hangup
Process start to fork.
I'm the Parent Process, my pid = 2753
I'm the Child Process, my pid = 2754
Child process start to execute test program.
------------CHILD PROCESS START------------
This is the SIGHUP program

Parent process receives SIGCHLD signal.
Child process raises the SIGHUP signal.
Hangup is detected in the terminal or death of controlling process.
```

- result of test case normal.c

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./normal
Process start to fork.
I'm the Parent Process, my pid = 2812
I'm the Child Process, my pid = 2813
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
```

- result of test case stop.c

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./stop
Process start to fork.
I'm the Parent Process, my pid = 2877
I'm the Child Process, my pid = 2878
Child process start to execute test program.
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receives SIGCHLD signal.
Parent process receives SIGSTOP signal.
The child process stopped.
```

- result of test case `illegal_instr.c`

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./illegal_instr
Process start to fork.
I'm the Parent Process, my pid = 1984
I'm the Child Process, my pid = 1985
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.
```

- result of test case interrupt.c

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./interrupt
Process start to fork.
I'm the Parent Process, my pid = 2040
I'm the Child Process, my pid = 2041
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.
```

- result of test case kill.c

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./kill
Process start to fork.
I'm the Parent Process, my pid = 2090
I'm the Child Process, my pid = 2091
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.
```

- result of test case pipe.c

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./pipe
Process start to fork.
I'm the Parent Process, my pid = 2139
I'm the Child Process, my pid = 2140
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.
```

- result of test case quit.c

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./quit
Process start to fork.
I'm the Parent Process, my pid = 2204
I'm the Child Process, my pid = 2205
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.
```

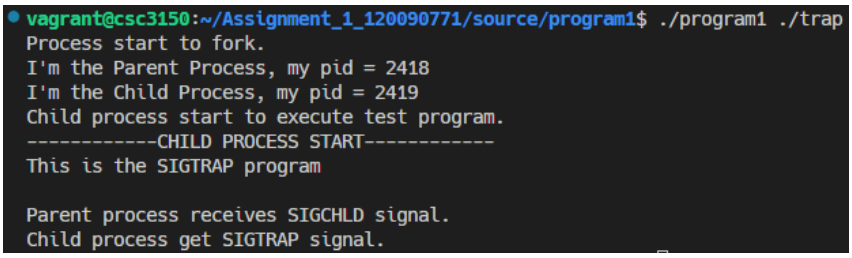
- result of test case `segment_fault.c`

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./segment_fault
Process start to fork.
I'm the Parent Process, my pid = 2293
I'm the Child Process, my pid = 2294
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.
```

- result of test case terminate.c

```
vagrant@csc3150:~/Assignment_1_120090771/source/program1$ ./program1 ./terminate
Process start to fork.
I'm the Parent Process, my pid = 2353
I'm the Child Process, my pid = 2354
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.
```

- result of test case trap.c



1.3 what I have learnt from task 1

In general, the most important thing in the task is to **quickly be familiar with how the related APIs are applied.** For example, what parameters are passed into the API and what is the return value of it.

In addition, **it is also important to learn how to coordinate different APIs such that the entire program work as expected.** For instance, when applying the `fork()`, there would be two return values since a new process has created. These two return values would be very useful in seperating the two processes. The first parameter of the API `waitpid()` is the PID of the child process, which can be obtained from the API `fork()`.

2 Task 2

The main task of task 2 is to implement a loadable kernel module and insert it into the base kernel. Inside the module, several tasks should be accomplished. When the module is initialized, a new **kernel thread** should be created. In the kernel thread, the `my_fork()` function is executed. In the `my_fork()` function, a new **process** is created. In the process, the test program is executed. The test process may raise signals making the child process terminates. In this case, the signal would be caught and print out in the signal log.

2.1 Design of the Program

Since the entire task is conducted in kernel mode, some of the functions used in the program may be implemented in the kernel source code as a static function. To apply these functions in the program, it is necessary to use `EXPORT_SYMBOL()` function to make these functions visible. In addition, once the kernel source code is modified, the kernel will need to be recompiled.

To create a kernel thread in kernel mode, the function `kthread_create()` is used in the initial module. Since the function `my_fork()` is executed in the kernel thread, the first parameter of `kthread_create()` should be a pointer point to the function `my_fork()`.

To fork a process in the `my_fork()` function, the function `kernel_clone()` is applied in the kernel mode. This function required a parameter `&args` which is a pointer to a kind of structure defined in the kernels source code called `kernel_clone_args`. To make this function fit with the case in task 2, attributes of the `kernel_clone_args` structure would be set as

```
.flag = SIGCHLD,
.exit_signal = SIGCHLD,
.stack = (unsigned long)&my_exec
```

The `my_exec()` function is the function that executes the test program in another file. By setting the above parameters, the child process may execute the `my_exec()` function as expected. In the `my_exec()` function, another function called `do_execve()` which is defined in the kernal source code could be applied to run the test program.

The signal from the child process should be received by the parent process. Therefore, in the previous `my_fork()` function, after invoking the function `kernel_clone`, the process needs to wait for the signals before its next move. Thus, the kernal mode function `do_wait()` is applied, whose parameter is `&wait_opts`, a pointer to a structure defined in the kernel source code file. Unlike other structures which is defined in the header file, this structure is defined in a .c file. Therefore, to use the structure, the prototype of the structure should be listed in program2.c before it is used. The return value of the function would be the PID of the child process, which is not the termination status of the child process. Instead, the attribute `wo_stat` of the struture `wait_opts` would record the information of the termination signals of the child process. Therefore, in the next step that dealing with the resulting signal, the value of this attribute would be useful.

Inspired by task 1, to deal with the signals, it is necessary to implement functions that work similarly as the functions mentioned in task 1 which can handle the signals. With all this functions, the corresponding signal code of a signal could be found and the related message could be printed out.

2.2 Environment Set Up (Kernel Compilation)

The system environment used in accomplishing task 2 is **linux-5.10.5**. To set up the environment, the source code of version 5.10.5 is downloaded from relevant websites. After that, the source code would be compiled in the original virtual machine environment. First of all, unzip the downloaded files by using the following command.

```
sudo tar xvf FILE_NAME.tar.xz
```

After that, the original kernel file is copied to the new kernel file. In order to compile the new version kernel file, the previous settings should be removed first by using the command:

```
make mrproper
make clean
```

Then, the configuration can be started by the command `make menuconfig`. Before installing the kernel and the kernel module, the kernel image and kernel modules should be built first. Therefore, the following commend in used.

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

To install the modules and kernel, the following command is used.

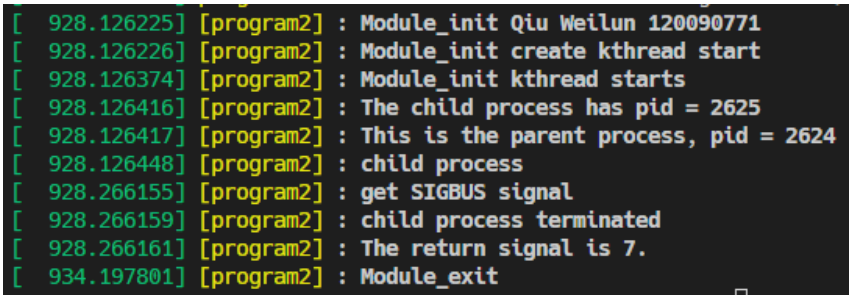
```
make modules_install
make install
```

After the installation and rebooting, the development environment is set up. The compilation process is similar to the process described above, except that it starts from the `make bzImage -j$(nproc)` step.

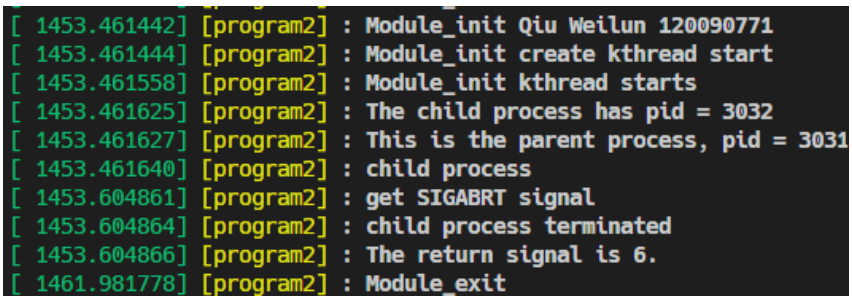
2.3 Program Outputs

The following snapshots are the outputs of some of the test cases.

- output of test program given in the folder "program2" (SIGBUS)



- output of execution using test file abort.c given in the folder "program1" (SIGABRT)



- output of excution using file alarm.c given in the folder "program1" (SIGALARM)

```
[ 1791.909457] [program2] : Module_init Qiu Weilun 120090771
[ 1791.909459] [program2] : Module_init create kthread start
[ 1791.909638] [program2] : Module_init kthread starts
[ 1791.909711] [program2] : The child process has pid = 3426
[ 1791.909712] [program2] : This is the parent process, pid = 3425
[ 1791.909731] [program2] : child process
[ 1793.915346] [program2] : get SIGALARM signal
[ 1793.915348] [program2] : child process terminated
[ 1793.915349] [program2] : The return signal is 14.
[ 1796.685023] [program2] : Module_exit
```

- output of excution using file floating.c given in the folder "program1" (SIGFPE)

```
[ 1965.973520] [program2] : Module_init Qiu Weilun 120090771
[ 1965.973522] [program2] : Module_init create kthread start
[ 1965.973673] [program2] : Module_init kthread starts
[ 1965.973792] [program2] : The child process has pid = 2917
[ 1965.973795] [program2] : This is the parent process, pid = 2916
[ 1965.973818] [program2] : child process
[ 1966.096971] [program2] : get SIGFPE signal
[ 1966.096974] [program2] : child process terminated
[ 1966.096975] [program2] : The return signal is 8.
[ 1970.845782] [program2] : Module_exit
```

- output of the execution using file `illegal_instr.c` given in the folder "program1" (SIGILL)

```
[ 2573.701471] [program2] : Module_init Qiu Weilun 120090771
[ 2573.701473] [program2] : Module_init create kthread start
[ 2573.701579] [program2] : Module_init kthread starts
[ 2573.701642] [program2] : The child process has pid = 3851
[ 2573.701643] [program2] : This is the parent process, pid = 3850
[ 2573.701649] [program2] : child process
[ 2573.828489] [program2] : get SIGILL signal
[ 2573.828493] [program2] : child process terminated
[ 2573.828494] [program2] : The return signal is 4.
[ 2577.373984] [program2] : Module_exit
```

- output of the execution using file `interrupt.c` given in the folder "program1" (SIGINT)

```
[ 3732.670886] [program2] : Module_init Qiu Weilun 120090771
[ 3732.670887] [program2] : Module_init create kthread start
[ 3732.671009] [program2] : Module_init kthread starts
[ 3732.671092] [program2] : The child process has pid = 4863
[ 3732.671094] [program2] : This is the parent process, pid = 4862
[ 3732.671097] [program2] : child process
[ 3732.680125] [program2] : get SIGINT signal
[ 3732.680127] [program2] : child process terminated
[ 3732.680128] [program2] : The return signal is 2.
[ 3739.941266] [program2] : Module_exit
```

- output of the execution using file `stop.c` given in the folder "program1" (SIGSTOP)

```
[ 4042.727349] [program2] : Module_init Qiu Weilun 120090771
[ 4042.727351] [program2] : Module_init create kthread start
[ 4042.727449] [program2] : Module_init kthread starts
[ 4042.727516] [program2] : The child process has pid = 5294
[ 4042.727517] [program2] : This is the parent process, pid = 5293
[ 4042.727525] [program2] : child process
[ 4042.736186] [program2] : CHILD PROCESS STOPPED
[ 4042.736188] [program2] : get SIGSTOP signal
[ 4042.736188] [program2] : the return signal is 19
[ 4046.501986] [program2] : Module_exit
```

- output of the execution using file `normal.c` given in the folder "program1" (normal termination)

```
[ 4502.486779] [program2] : Module_init Qiu Weilun 120090771
[ 4502.511646] [program2] : Module_init create kthread start
[ 4502.542367] [program2] : Module_init kthread starts
[ 4502.563562] [program2] : The child process has pid = 6492
[ 4502.563581] [program2] : child process
[ 4502.565861] [program2] : This is the parent process, pid = 6491
[ 4502.583454] [program2] : child process terminated
[ 4502.609280] [program2] : normal termination with EXIT STATUS = 0
[ 4509.622147] [program2] : Module_exit
```

2.4 what I have learnt from task 2

The most important thing I have learnt in the task is that **when designing a program, previous program with similar motivation may bring some useful ideas on the design of new one.** The task in task 2 is actually similar to task 1. The difference of user mode and kernel mode makes the methods available for the two tasks different. For instance, to fork a process, the `fork()` function is available for task 1. However, it is not available in kernel mode. Therefore, to create a new process in task 2, method with lower level abstraction are needed. Thus, the `kernel_clone()` function would be an alternative for the original `fork()` function. In fact, the implmentation of the `fork()` function is based on the function `kernel_clone()` in the linux souce code. Similarly, in both task 1 and task 2, the parent process should wait for the child process to terminate and deliver signals. In task 1, the function `wait_pid()` is applied. The alternative function in task 2 would be `do_wait()`. The value of attribute `wo_stat` is equivalent to the return value of `wait_pid` function, both of which keep the information of child process termination signal. In general, when designing the program in task 2, the implementation of task 1 could be a reference to the design of task 2.

In addition, **higher level of abstraction may guarantee security.** In task 1, even if the program does not executed as expected, the entire operating system kernel would not crash because of the protection machanism in lower level function. However, in task 2, the kernel may crashed since a loadable kernel module is inserted into the kernel and the entire process is conducted under kernel mode.

Finally, **when programing under kernel mode, it is important to check the kernel source code before modifying a new module.** Reading the source code would

be a direct way to learn about what would happen when the corresponding function is invoked. More impotantly, it would be a safer way to modify the kernel after reading and understanding the kernel source code.

3 Bonus

In linux system, the command `pstree` provides users with a way to print out the process tree. In the bonus task, the program is designed to have similar functionality.

3.1 Design of the Program

In the ubuntu virtual machine, the information of all the processes are stored in the following directory.

```
/proc/PROCESS_ID/status
```

Therefore, in the program, the directory would be first opened and read. After that, the related information such as process ID could be collected from the files.

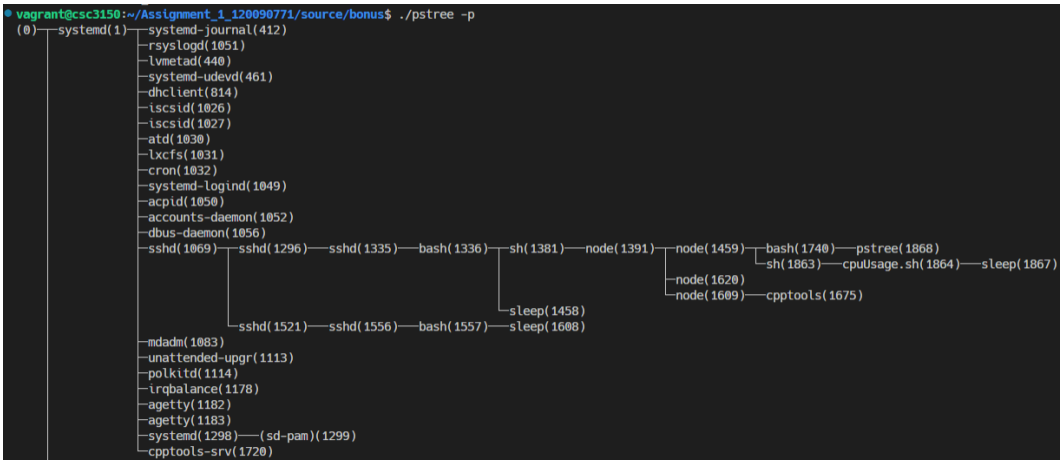
Since the output of the command is a tree structure. It is reasonable to use linked list data structure to implement the entire tree structure. After the tree is constructed, the tree could be printed out recursively.

The program also support the option "-p" in the `pstree` command. To implmment the "-p", the process IDs are collected in the files as mentioned before. When the tree is printed out, not only the name of the process but also the process ID are printed out.

3.2 Output of the Program

The following figure shows part of the output process tree.

- output of command `./pstree -p`:



- output of command `./pstree`:

3.3 Make File for Bonus

A make file is included in the bonus part. Before running the program, please enter the command `make` to compile the source code in `pstree.c`. To clear the objective file, please use the command `make clean`.

4 About Coding Style

The `.clang-format` file in each folder is used to format the code. It is a copy of the `.format-format` file in the kernel source code file of linux-5.10.5. All the code in the program has already formatted before submission. If the format is inconsistent with the style the 5.10.5 version kernel source code, please use the following command to format the code.

```
clang-format -i FILE_NAME.c
```

Notice that the `.clang-format` file works well with clang-format version higher than clang-format-4.0. Therefore, please make sure that the version is correct.

5 Notice

- The program outputs of task 1 shown above is invalid due to the change of the assignment requirement. To see the correct output, please run the source code in `program1.c`.