Assignment Report

CSC 3150 Kernel-Mode Multi-Process Programming

Wei Wu (吴畏)

118010335

October 3, 2020

The School of Data Science

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

1. How Did I Design My Program

1.1. Program 1

This program forks a child process to execute the test program. When the child process finishes execution, it will send a SIGCHLD signal to the parent process, while the parent process will receive this signal by wait() function. In the end, the termination information will be printed out.

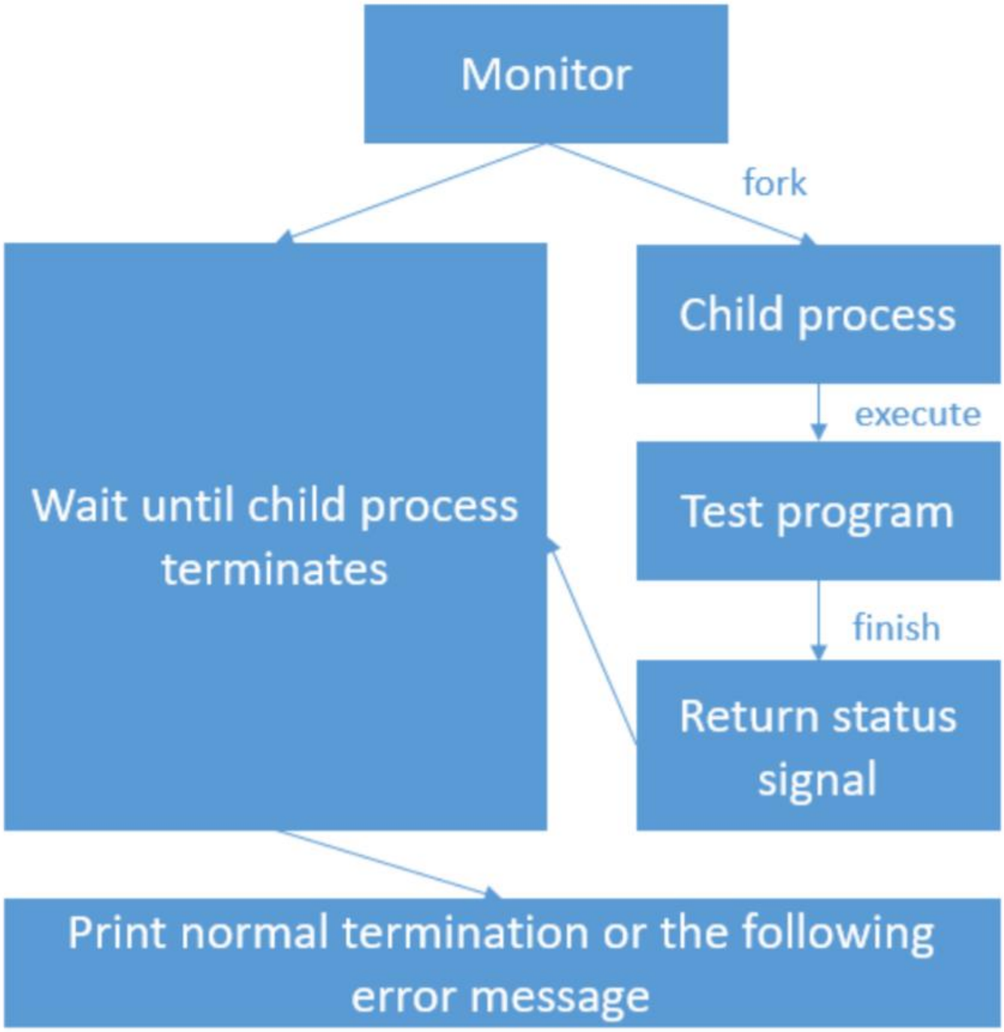
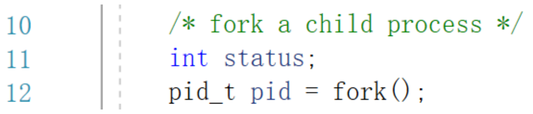

Monitor
fork
Child process
execute
Test program
finish
Return status signal
Wait until child process terminates
Print normal termination or the following error message

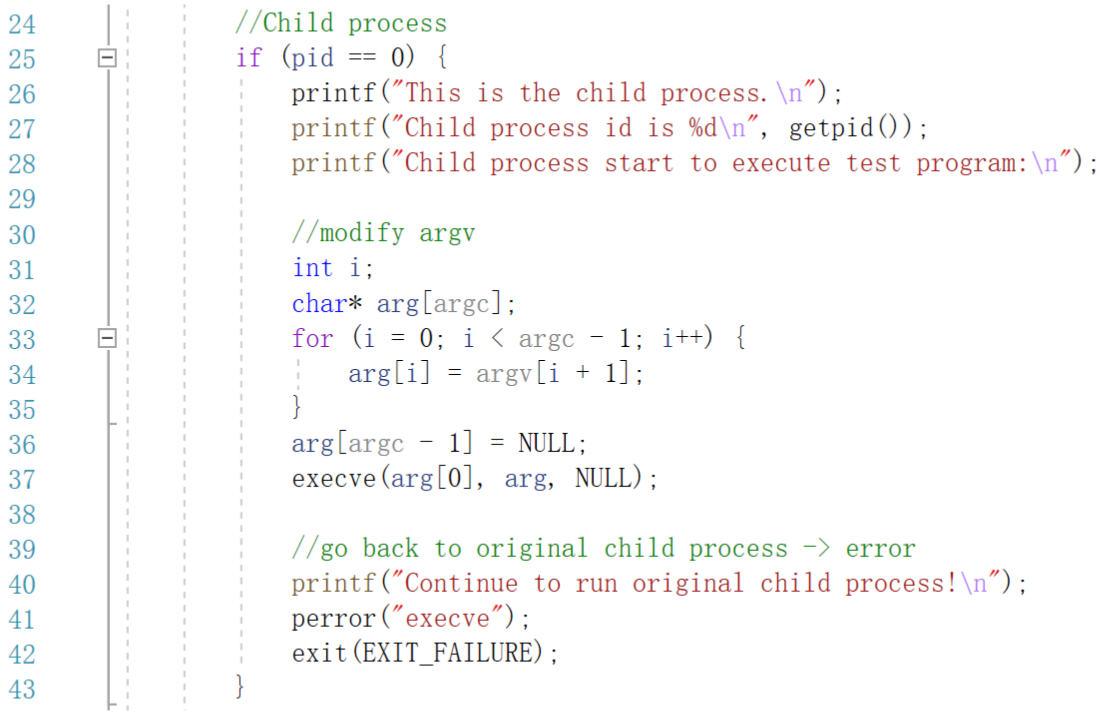
Figure 1. ***The Main Flow Chart of Program 1***

a. Fork a child process using fork()



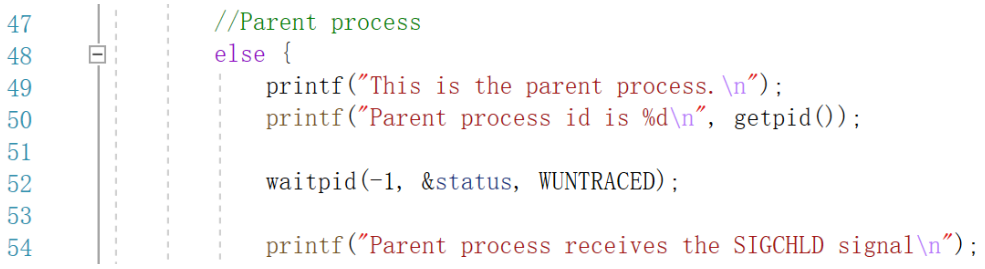
```
/* fork a child process */
int status;
pid_t pid = fork();
```

b. If fork() succeeds, the child process will execute the test program.



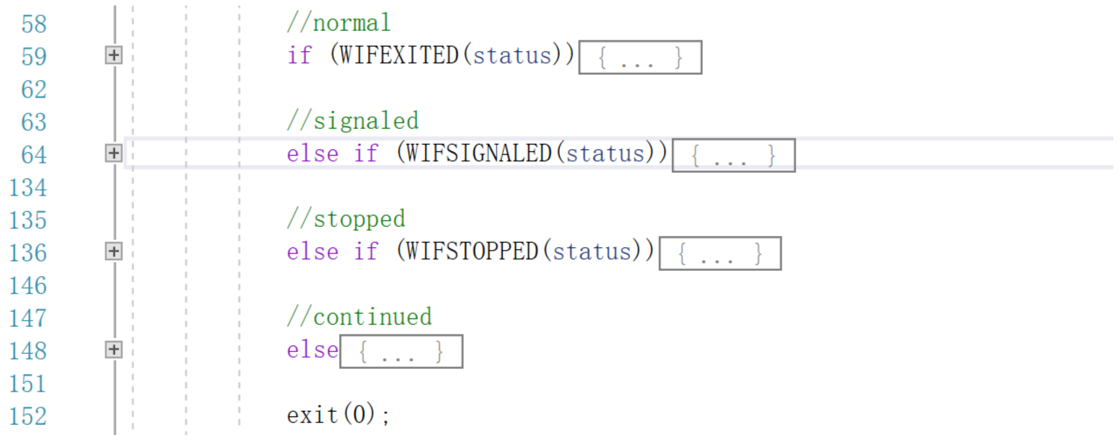
```
//Child process
if (pid == 0) {
    printf("This is the child process.\n");
    printf("Child process id is %d\n", getpid());
    printf("Child process start to execute test program:\n");

    //modify argv
    int i;
    char* arg[argc];
    for (i = 0; i < argc - 1; i++) {
        arg[i] = argv[i + 1];
    }
    arg[argc - 1] = NULL;
    execve(arg[0], arg, NULL);

    //go back to original child process -> error
    printf("Continue to run original child process!\n");
    perror("execve");
    exit(EXIT_FAILURE);
}
```

c. If fork() succeeds, the parent process will wait until the child process terminates.



```
//Parent process
else {
    printf("This is the parent process.\n");
    printf("Parent process id is %d\n", getpid());

    waitpid(-1, &status, WUNTRACED);

    printf("Parent process receives the SIGCHLD signal\n");
```

d. When the parent process receives the SIGCHLD signal, it will check and print the child process' termination status.



```
//normal
if (WIFEXITED(status)){ ... }

//signaled
else if (WIFSIGNALED(status)){ ... }

//stopped
else if (WIFSTOPPED(status)){ ... }

//continued
else{ ... }

exit(0);
```

e. The program terminates.

1.2. Program 2

This program is basically a kernel object which forks a process to execute the test program. When initialized, it creates a kernel thread and run my_fork function. This function forks a child process to execute the test program and makes the parent process to wait until the child process terminates. Meanwhile, it prints out the process ID of both parent and child process. In the test program, a signal will be raised. This signal will be received by the parent process, and then the related message will be printed out in the kernel log.

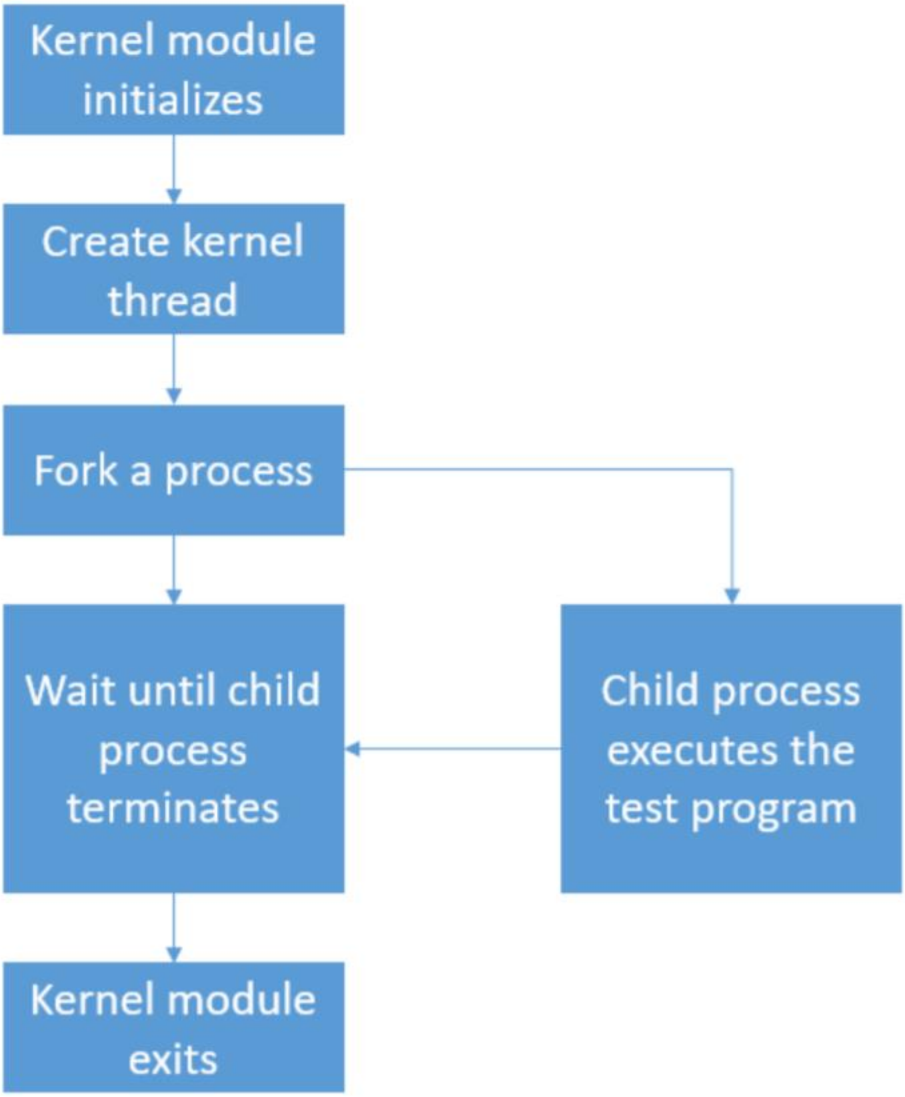
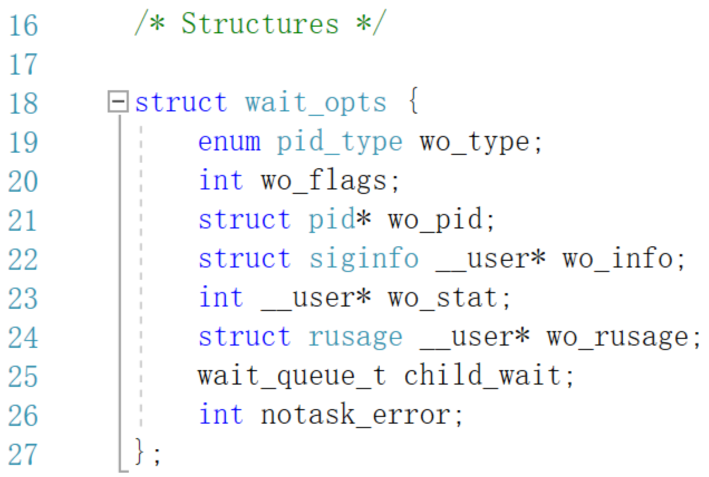

Kernel module initializes
Create kernel thread
Fork a process
Wait until child process terminates
Child process executes the test program
Kernel module exits

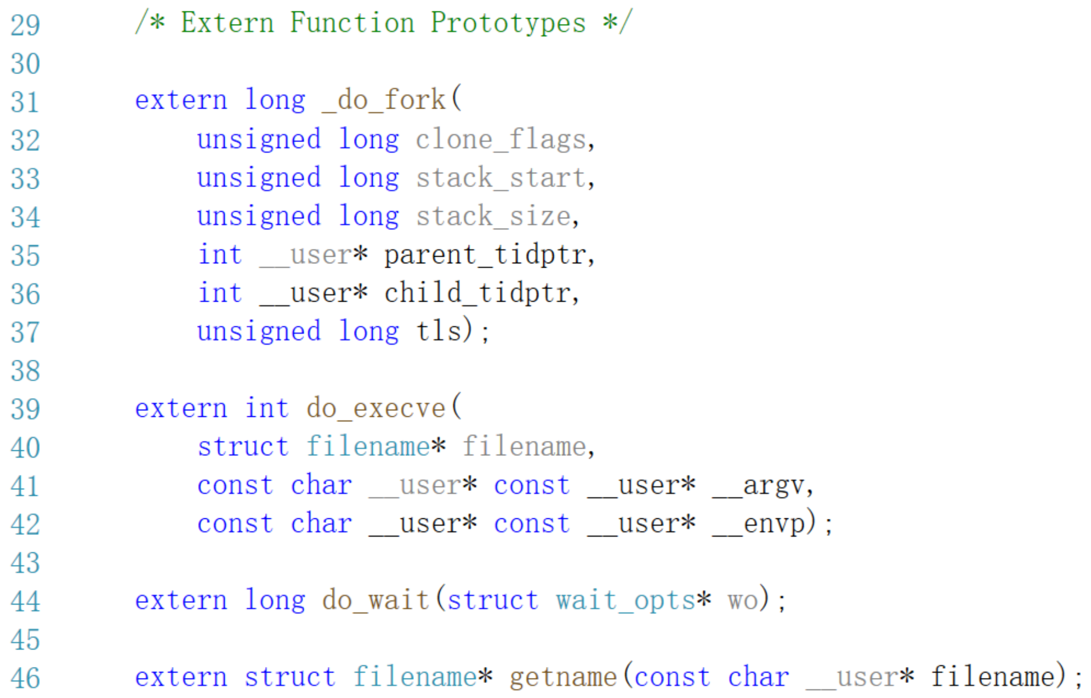
Figure 2. ***The Main Flow Chart of Program 2***

a. Declare the wait_opts struct.



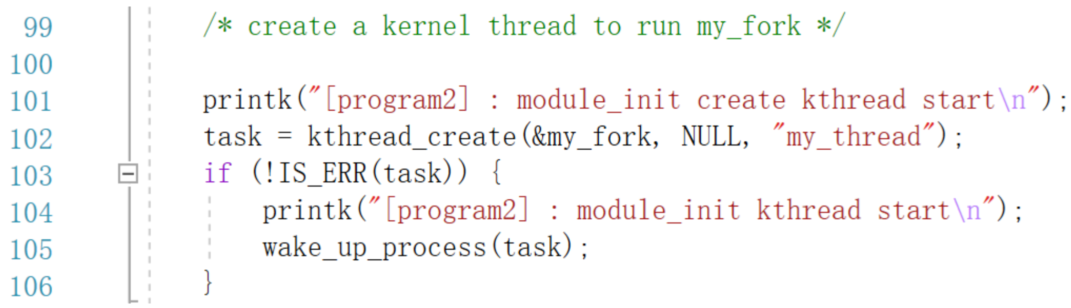
```
/* Structures */

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;
};
```

b. Declare the extern functions



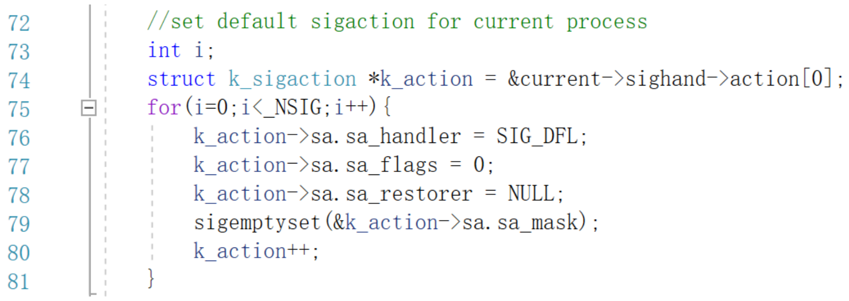
```
/* Extern Function Prototypes */

extern long _do_fork(
    unsigned long clone_flags,
    unsigned long stack_start,
    unsigned long stack_size,
    int __user* parent_tidptr,
    int __user* child_tidptr,
    unsigned long tls);

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

extern long do_wait(struct wait_opts* wo);

extern struct filename* getname(const char __user* filename);
```

c. When initialized, create a kernel thread to run my_fork()



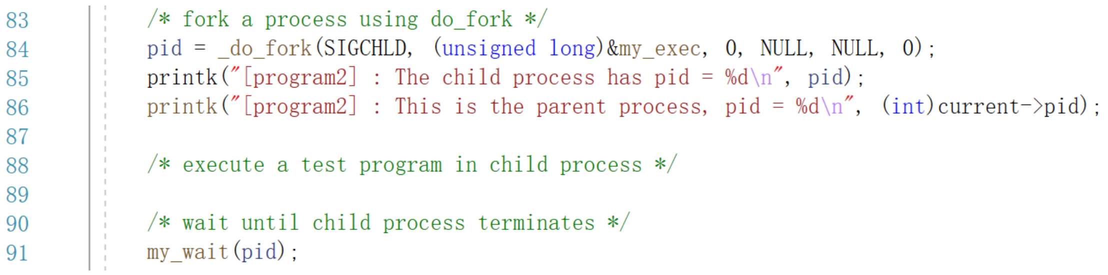
```
/* create a kernel thread to run my_fork */

printk("[program2] : module_init create kthread start\n");
task = kthread_create(&my_fork, NULL, "my_thread");
if (!IS_ERR(task)) {
    printk("[program2] : module_init kthread start\n");
    wake_up_process(task);
}
```

d. In my_fork(), first set default sigaction for the current process.



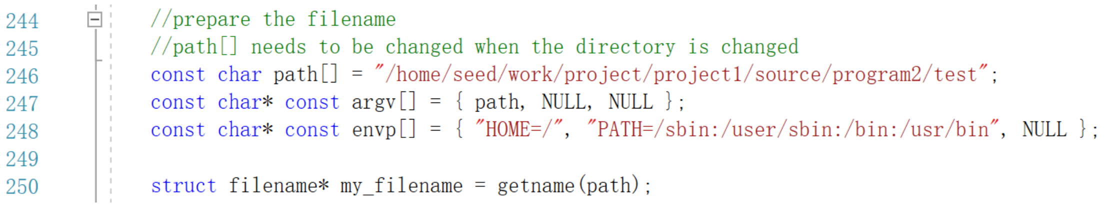
```
//set default sigaction for current process
int i;
struct k_sigaction *k_action = &current->sighand->action[0];
for(i=0;i<_NSIG;i++){
    k_action->sa.sa_handler = SIG_DFL;
    k_action->sa.sa_flags = 0;
    k_action->sa.sa_restorer = NULL;
    sigemptyset(&k_action->sa.sa_mask);
    k_action++;
}
```

e. Then fork a child process to execute the test program using do_fork() and my_exec(). Then call my_wait() to make the parent process wait until the child process terminates.



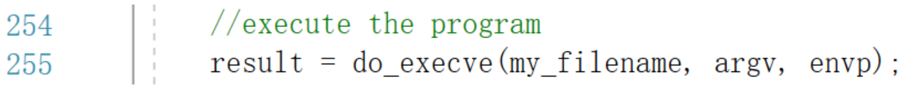
```
/* fork a process using do_fork */
pid = _do_fork(SIGCHLD, (unsigned long)&my_exec, 0, NULL, NULL, 0);
printk("[program2] : The child process has pid = %d\n", pid);
printk("[program2] : This is the parent process, pid = %d\n", (int)current->pid);

/* execute a test program in child process */

/* wait until child process terminates */
my_wait(pid);
```

f. In my_exec(), first prepare the arguments for the do_execve() function.



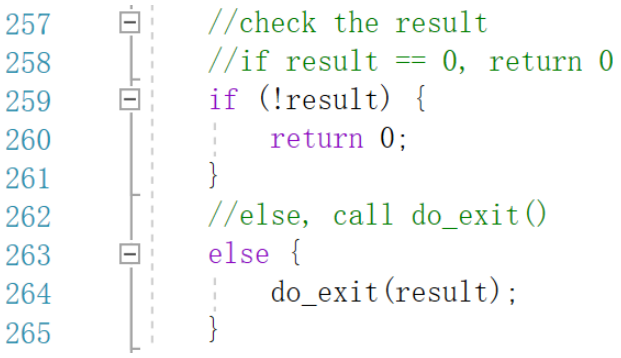
```
//prepare the filename
//path[] needs to be changed when the directory is changed
const char path[] = "/home/seed/work/project/project1/source/program2/test";
const char* const argv[] = { path, NULL, NULL };
const char* const envp[] = { "HOME=/", "PATH=/sbin:/user/sbin:/bin:/usr/bin", NULL };

struct filename* my_filename = getname(path);
```

g. Then call do_execve() to execute the program.



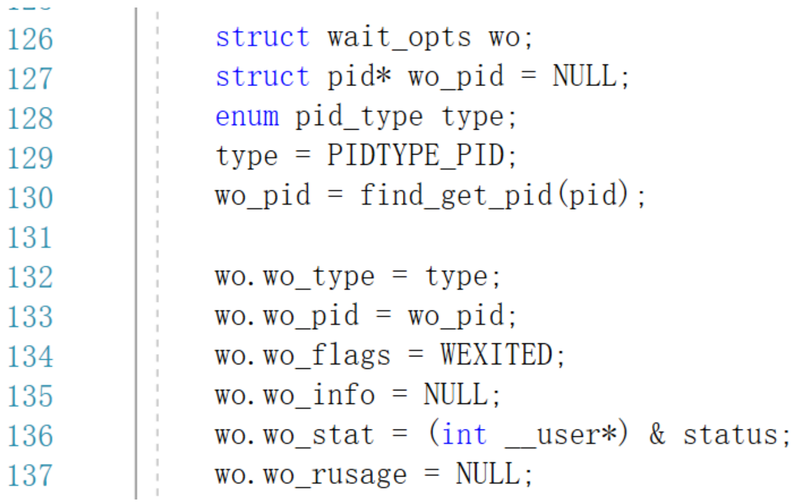
```
//execute the program
result = do_execve(my_filename, argv, envp);
```

h. If the result is 0, return 0 (normal termination). Otherwise call do_exit() to deal with the exception.



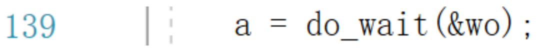
```
//check the result
//if result == 0, return 0
if (!result) {
    return 0;
}
//else, call do_exit()
else {
    do_exit(result);
}
```

i. In my_wait(), first create and initialize a wait_opts struct.



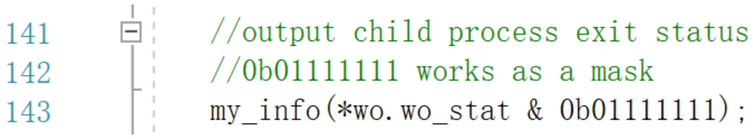
```
struct wait_opts wo;
struct pid* wo_pid = NULL;
enum pid_type type;
type = PIDTYPE_PID;
wo_pid = find_get_pid(pid);

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;
```

j. Call do_wait() to make the parent process wait.



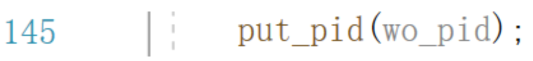
```
a = do_wait(&wo);
```

k. Call my_info() to output the child process’ termination information.



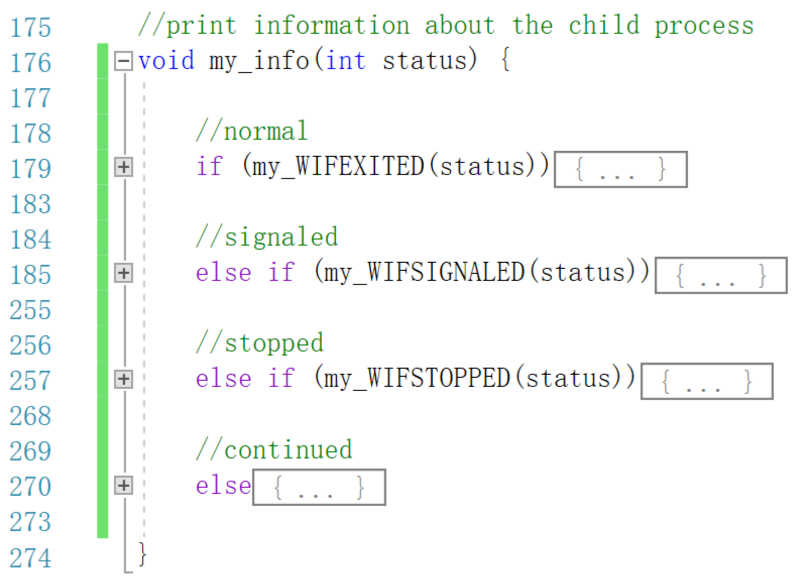
```
//output child process exit status
//0b01111111 works as a mask
my_info(*wo.wo_stat & 0b01111111);
```

l. Call put_pid() to decrease the count of wo_pid in the hash table and free the memory allocated.



```
put_pid(wo_pid);
```

m. In my_info(), print out the termination information about the child process.



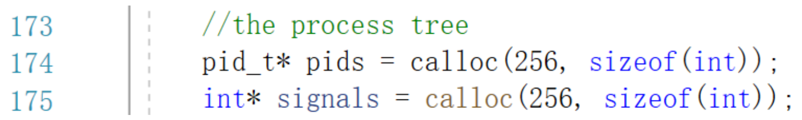
```
//print information about the child process
void my_info(int status) {

    //normal
    if (my_WIFEXITED(status)) { ... }

    //signaled
    else if (my_WIFSIGNALED(status)) { ... }

    //stopped
    else if (my_WIFSTOPPED(status)) { ... }

    //continued
    else { ... }
}
```

n. The program terminates.

1.3. Bonus

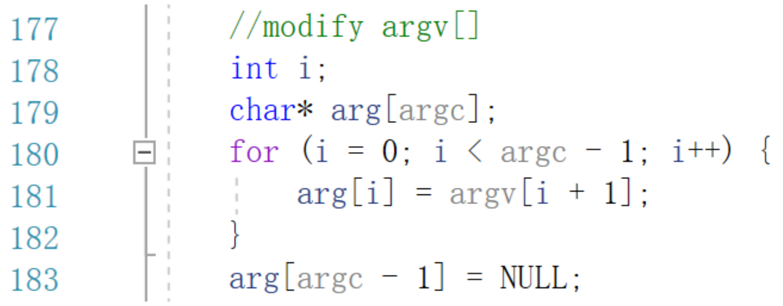
This program recursively forks a process to execute a series of programs. Besides, it prints out a process tree which indicates the relationship of the test programs. It also prints out the termination information of each process.

a. Allocate memory for the process tree.



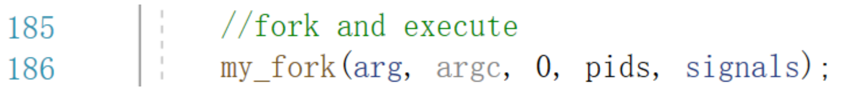
```
    //the process tree
    pid_t* pids = calloc(256, sizeof(int));
    int* signals = calloc(256, sizeof(int));
```

b. Modify argv[].



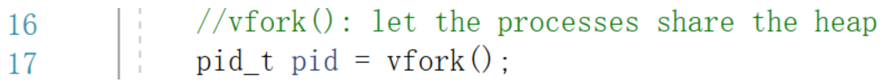
```
    //modify argv[]
    int i;
    char* arg[argc];
    for (i = 0; i < argc - 1; i++) {
        arg[i] = argv[i + 1];
    }
    arg[argc - 1] = NULL;
```

c. Call my_fork() to fork processes and execute programs recursively.



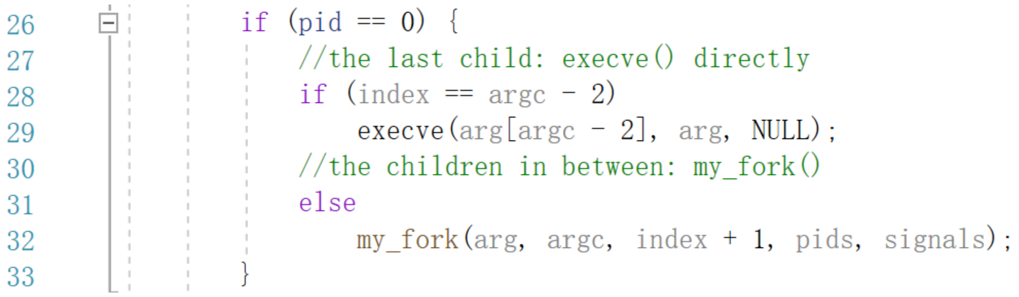
```
//fork and execute
my_fork(arg, argc, 0, pids, signals);
```

d. In my_fork(), use vfork() to let the processes share the heap storage.



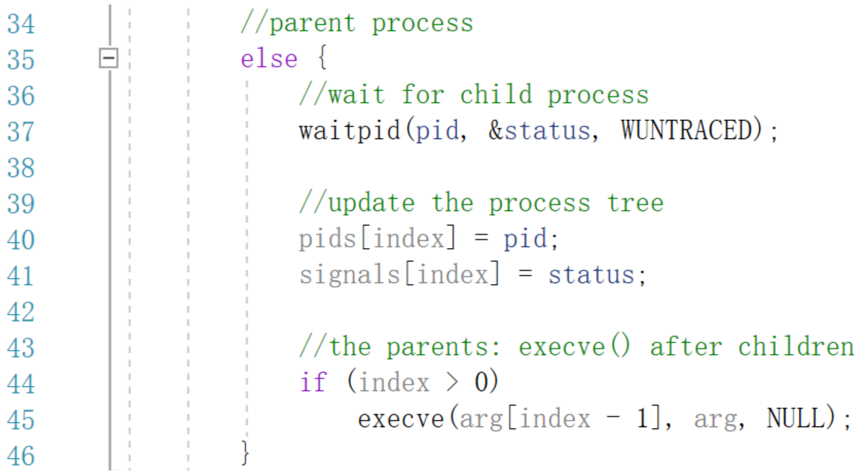
```
//vfork(): let the processes share the heap
pid_t pid = vfork();
```

e. If no fork error, the last child call execve() directly while the other children call my_fork() recursively.



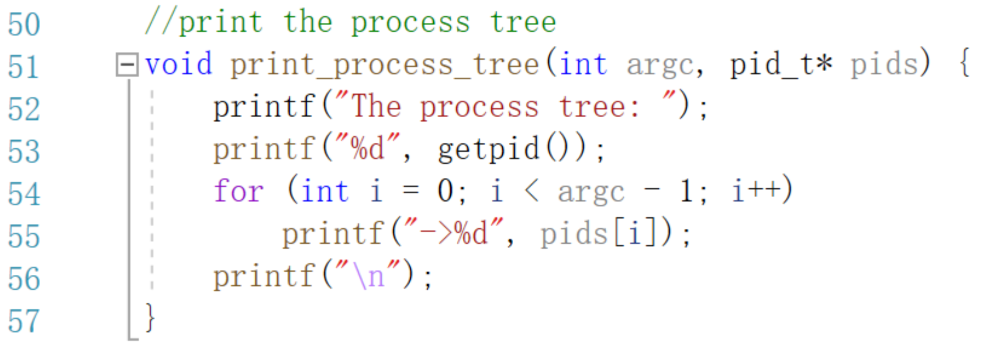
```
if (pid == 0) {
    //the last child: execve() directly
    if (index == argc - 2)
        execve(arg[argc - 2], arg, NULL);
    //the children in between: my_fork()
    else
        my_fork(arg, argc, index + 1, pids, signals);
}
```

f. The parent processes must wait for their corresponding child process' termination. After that, they update the process tree with the child process' process ID and termination status. Then, the parent processes, except the first process, call execve() to execute the programs.



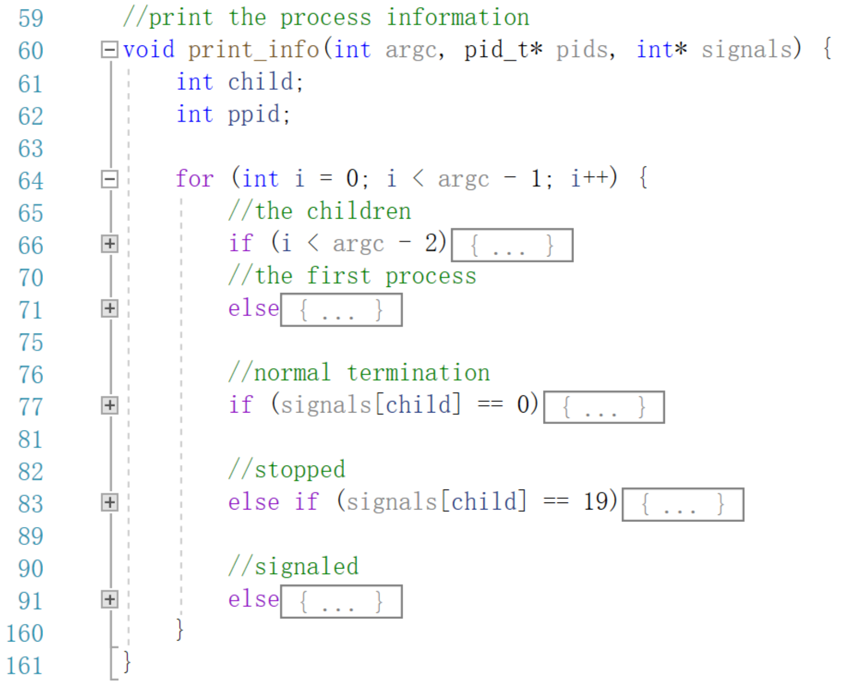
```
//parent process
else {
    //wait for child process
    waitpid(pid, &status, WUNTRACED);

    //update the process tree
    pids[index] = pid;
    signals[index] = status;

    //the parents: execve() after children
    if (index > 0)
        execve(arg[index - 1], arg, NULL);
}
```

g. Then, main() calls print_process_tree() to print the process tree.



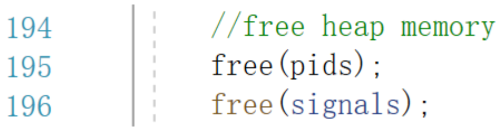
```
    //print the process tree
void print_process_tree(int argc, pid_t* pids) {
    printf("The process tree: ");
    printf("%d", getpid());
    for (int i = 0; i < argc - 1; i++)
        printf("->%d", pids[i]);
    printf("\n");
}
```

h. Also, main() calls print_info() to print the process information.



```
    //print the process information
void print_info(int argc, pid_t* pids, int* signals) {
    int child;
    int ppid;

    for (int i = 0; i < argc - 1; i++) {
        //the children
        if (i < argc - 2){ ... }
        //the first process
        else{ ... }

        //normal termination
        if (signals[child] == 0){ ... }

        //stopped
        else if (signals[child] == 19){ ... }

        //signaled
        else{ ... }
    }
}
```

i. In the end, main() frees the heap memory allocated.



```
    //free heap memory
    free(pids);
    free(signals);
```

j. The program terminates.

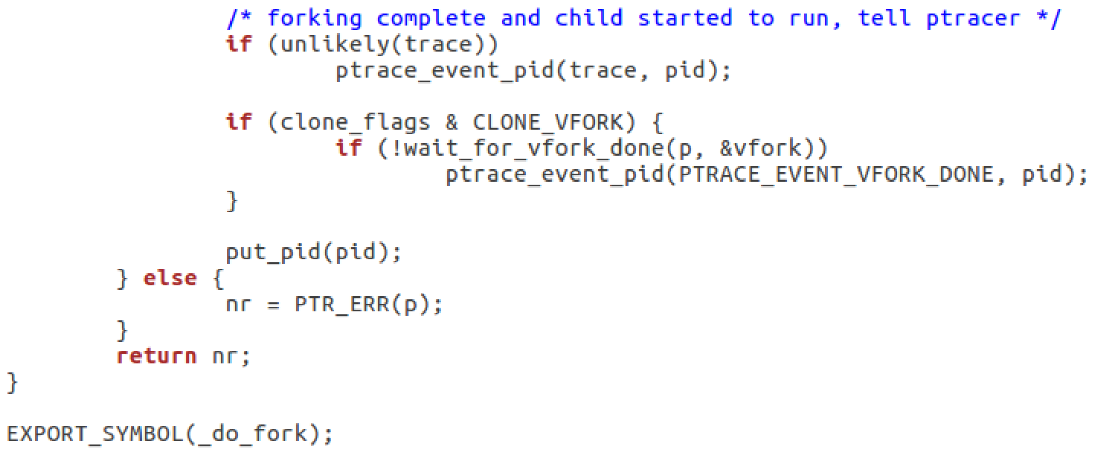
2. The Environment of Running My Program

OS: Linux version 4.10.14 (root@VM) (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.4))

Kernel: 4.10.14 with modification

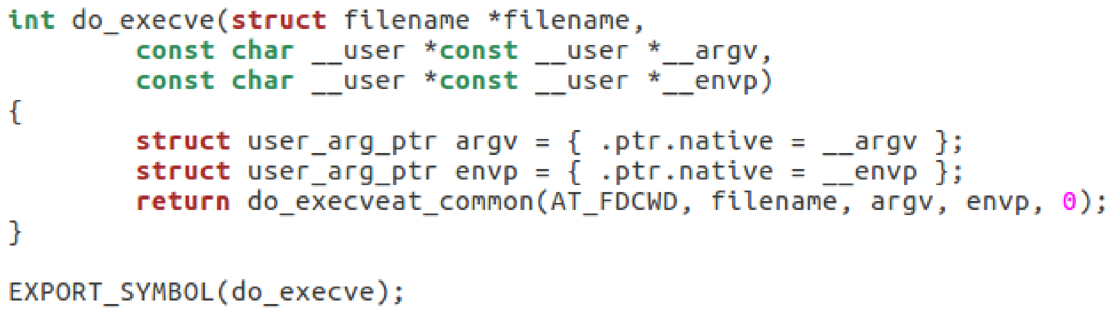
Kernel Modification:

a. do_fork() (/kernel/fork.c)



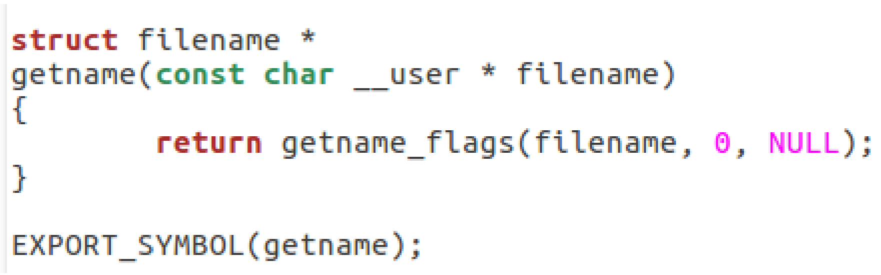
```
                /* forking complete and child started to run, tell ptracer */
                if (unlikely(trace))
                        ptrace_event_pid(trace, pid);

                if (clone_flags & CLONE_VFORK) {
                        if (!wait_for_vfork_done(p, &vfork))
                                ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
                }

                put_pid(pid);
        } else {
                nr = PTR_ERR(p);
        }
        return nr;
}

EXPORT_SYMBOL(_do_fork);
```

b. do_execve() (/fs/exec.c)



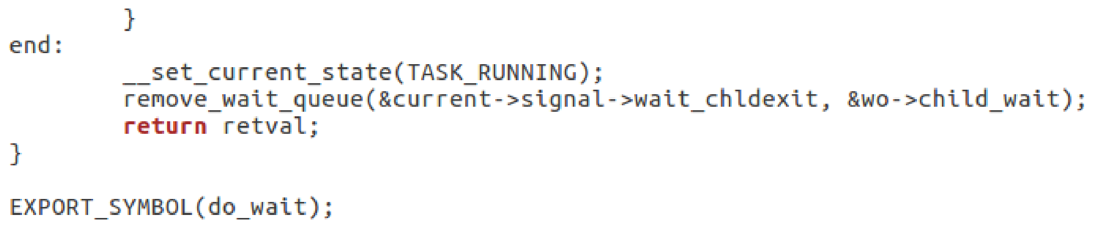
```
int do_execve(struct filename *filename,
        const char __user *const __user *__argv,
        const char __user *const __user *__envp)
{
        struct user_arg_ptr argv = { .ptr.native = __argv };
        struct user_arg_ptr envp = { .ptr.native = __envp };
        return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
}

EXPORT_SYMBOL(do_execve);
```

c. getname() (/fs/namei.c)



```
struct filename *
getname(const char __user * filename)
{
        return getname_flags(filename, 0, NULL);
}

EXPORT_SYMBOL(getname);
```

d. do_wait() (/kernel/exit.c)



```
        }
end:
        __set_current_state(TASK_RUNNING);
        remove_wait_queue(&current->signal->wait_chldexit, &wo->child_wait);
        return retval;
}

EXPORT_SYMBOL(do_wait);
```

3. The Steps to Execute My Program

3.1. Program 1

Under the 'program1' directory lies all source codes of Task 1 and test cases.

The 'program1.c' is the main program, the others are for test uses.

The directory consists of the following files:

program1.c, Makefile, abort.c, alarm.c, bus.c, floating.c, hangup.c, illegal_instr.c, interrupt.c, kill.c, normal.c, pipe.c, quit.c, segment_fault.c, stop.c, terminate.c, trap.c.

HOW TO COMPILE:

In the 'program1' directory, type 'make' command and enter.

HOW TO CLEAR:

In the 'program1' directory, type 'make clean' command and enter.

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.

3.2. Program 2

Under the 'program2' directory lies all source codes of Task 2 and one test case.

The 'program2.c' is the main program, and 'test.c' is for test use.

BEFORE PROGRAM COMPILATION AND EXECUTION:

Revising Linux Kernel is needed, as is shown in the following steps:

1. Update the Linux source code.

2. Compile the kernel and boot image, replace the boot image with new one, then reboot.

To compile to test program, simply type 'gcc -o $FILENAME $FILENAME.c', where

$FILENAME is the file name without the extension.

HOW TO COMPILE:

In the 'program2' directory, type 'make' command and enter

HOW TO CLEAR:

In the 'program2' directory, type 'make clean' command and enter.

HOW TO EXECUTE:

1.Type 'sudo insmod program2.ko' under 'program2' directory and enter

2.You could see messages appear by typing 'dmesg' command

The messages are between the messages 'module init' and 'module exit'.

3.Type 'sudo rmmod program2' and enter to remove the program2 module.

3.3. Bonus

Under the 'bonus' directory lies all source codes of bonus tasks and several test cases. The 'myfork.c' is the main program. Other *.c files are test cases.

HOW TO COMPILE:

In the 'bonus' directory, type 'make' command and enter.

Test programs could be compiled as well.

HOW TO CLEAR:

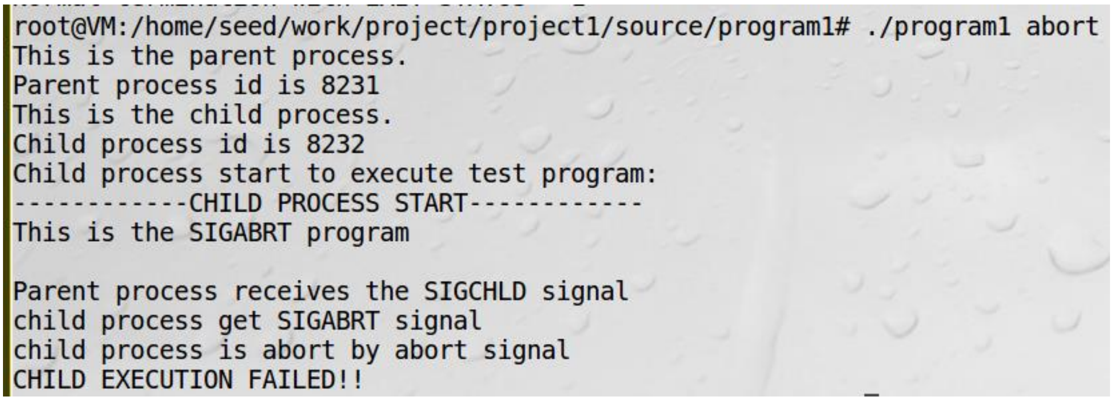
In the 'bonus' directory, type 'make clean' command and enter.

HOW TO EXECUTE:

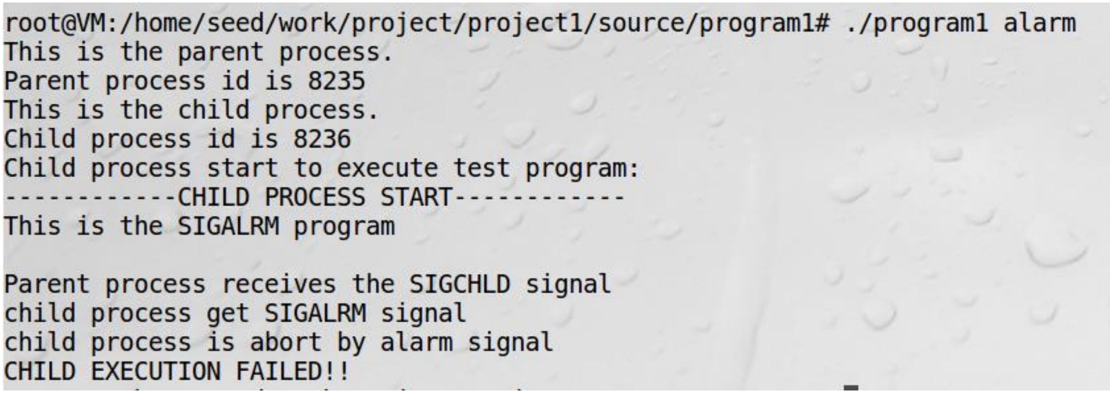
In the 'bonus' directory, type './myfork $TEST_PRO1 $TEST_PRO2 $TEST_PRO3 ...', where $TEST_PRO1, $TEST_PRO2,... are names of programs myfork executes.

4. Screenshots of My Program Output

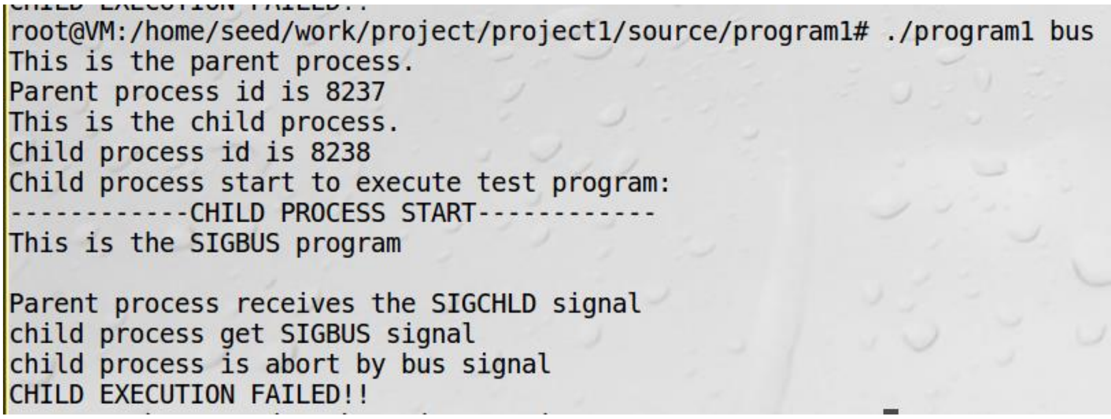
4.1. Program 1



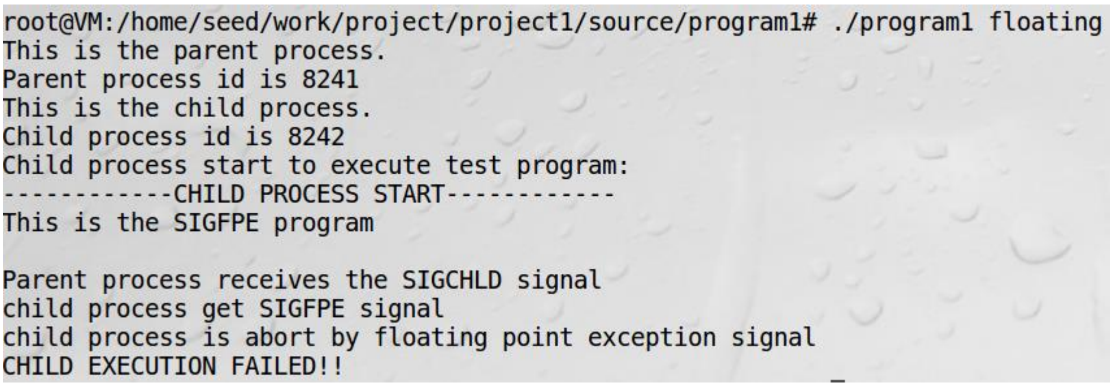
```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 abort
This is the parent process.
Parent process id is 8231
This is the child process.
Child process id is 8232
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGABRT program

Parent process receives the SIGCHLD signal
child process get SIGABRT signal
child process is abort by abort signal
CHILD EXECUTION FAILED!!
```



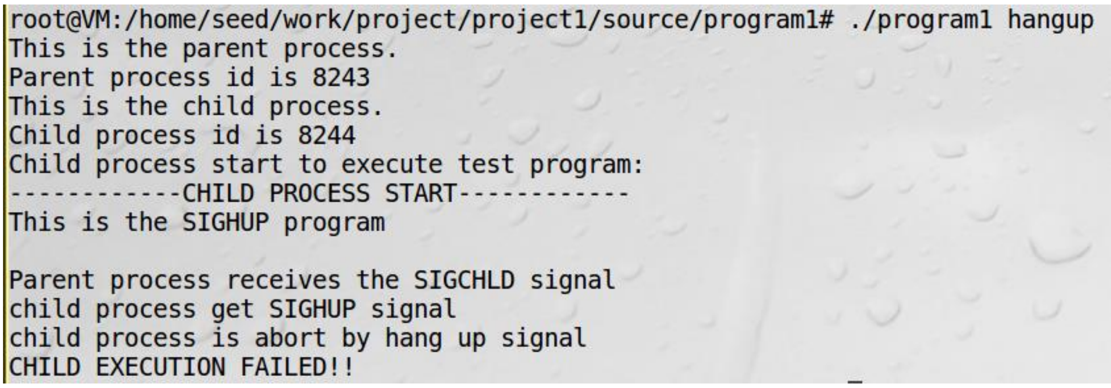
```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 alarm
This is the parent process.
Parent process id is 8235
This is the child process.
Child process id is 8236
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGALRM program

Parent process receives the SIGCHLD signal
child process get SIGALRM signal
child process is abort by alarm signal
CHILD EXECUTION FAILED!!
```



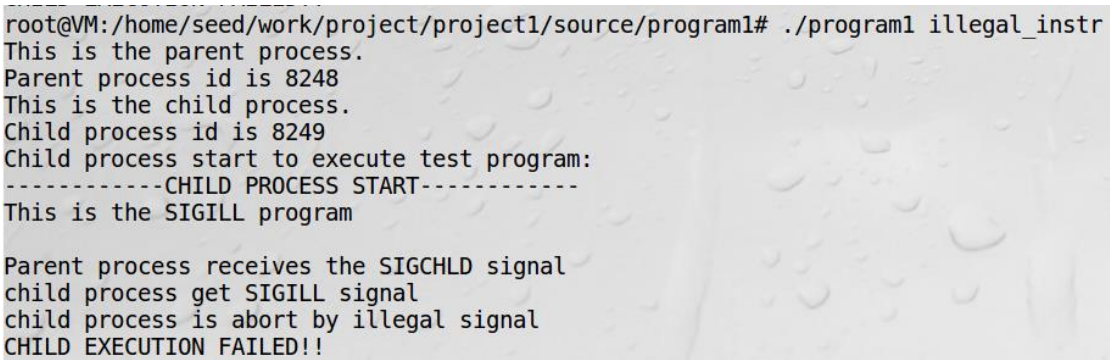
```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 bus
This is the parent process.
Parent process id is 8237
This is the child process.
Child process id is 8238
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGBUS program

Parent process receives the SIGCHLD signal
child process get SIGBUS signal
child process is abort by bus signal
CHILD EXECUTION FAILED!!
```



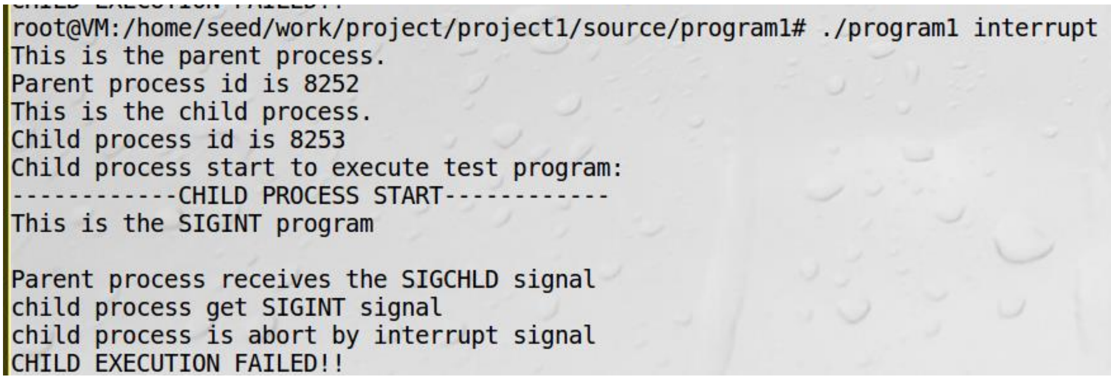
```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 floating
This is the parent process.
Parent process id is 8241
This is the child process.
Child process id is 8242
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGFPE program

Parent process receives the SIGCHLD signal
child process get SIGFPE signal
child process is abort by floating point exception signal
CHILD EXECUTION FAILED!!
```



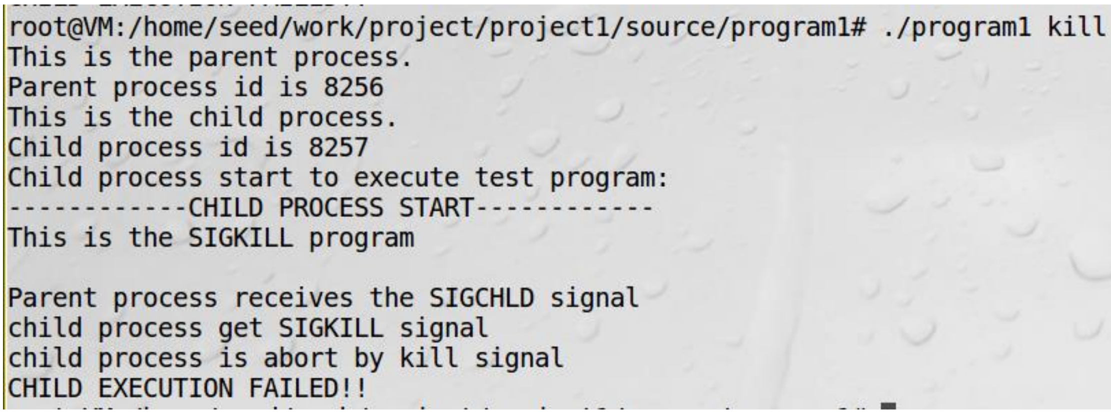
```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 hangup
This is the parent process.
Parent process id is 8243
This is the child process.
Child process id is 8244
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGHUP program

Parent process receives the SIGCHLD signal
child process get SIGHUP signal
child process is abort by hang up signal
CHILD EXECUTION FAILED!!
```



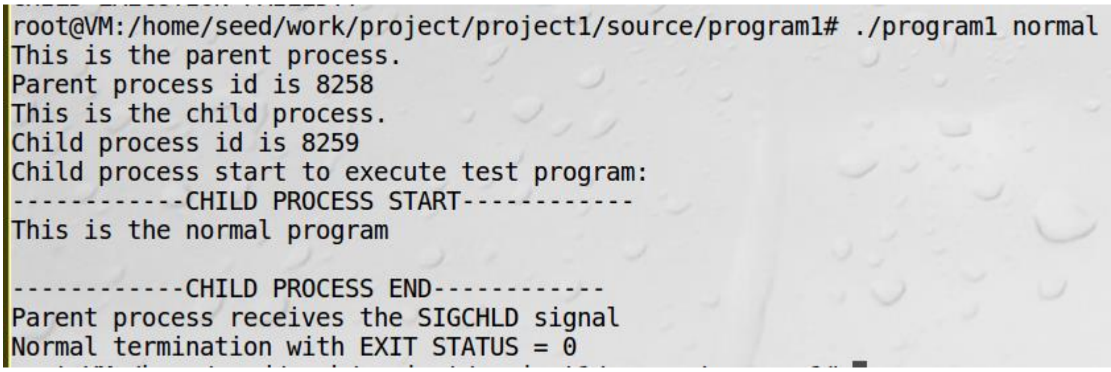
```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 illegal_instr
This is the parent process.
Parent process id is 8248
This is the child process.
Child process id is 8249
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGILL program

Parent process receives the SIGCHLD signal
child process get SIGILL signal
child process is abort by illegal signal
CHILD EXECUTION FAILED!!
```



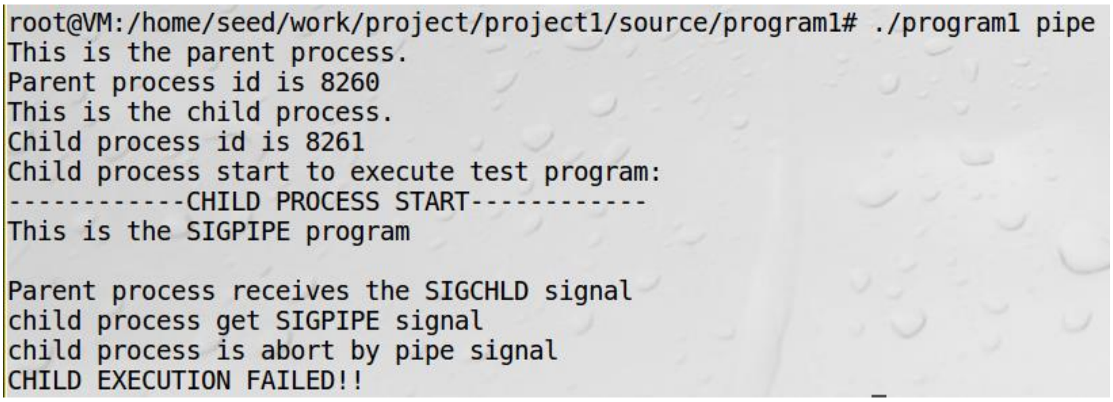
```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 interrupt
This is the parent process.
Parent process id is 8252
This is the child process.
Child process id is 8253
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGINT program

Parent process receives the SIGCHLD signal
child process get SIGINT signal
child process is abort by interrupt signal
CHILD EXECUTION FAILED!!
```



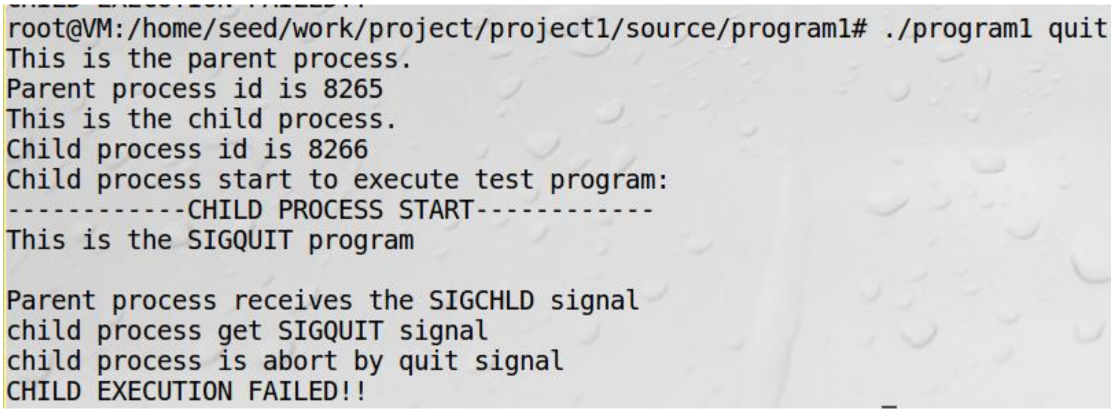
```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 kill
This is the parent process.
Parent process id is 8256
This is the child process.
Child process id is 8257
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGKILL program

Parent process receives the SIGCHLD signal
child process get SIGKILL signal
child process is abort by kill signal
CHILD EXECUTION FAILED!!
```



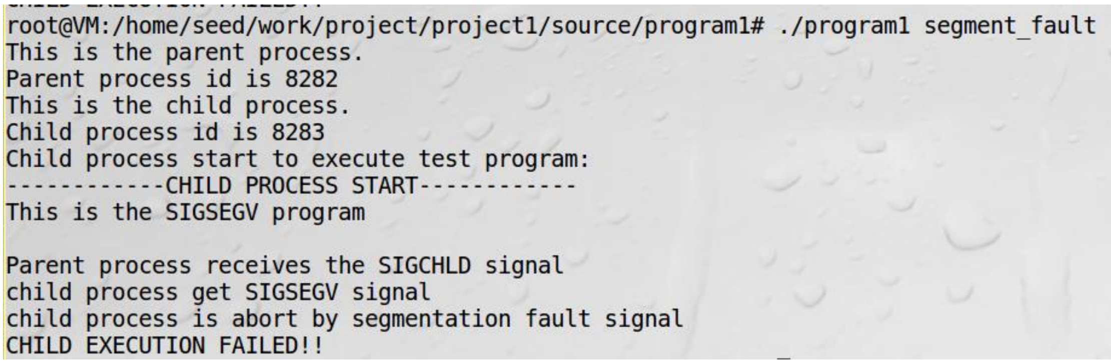
```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 normal
This is the parent process.
Parent process id is 8258
This is the child process.
Child process id is 8259
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the normal program

------------CHILD PROCESS END------------
Parent process receives the SIGCHLD signal
Normal termination with EXIT STATUS = 0
```



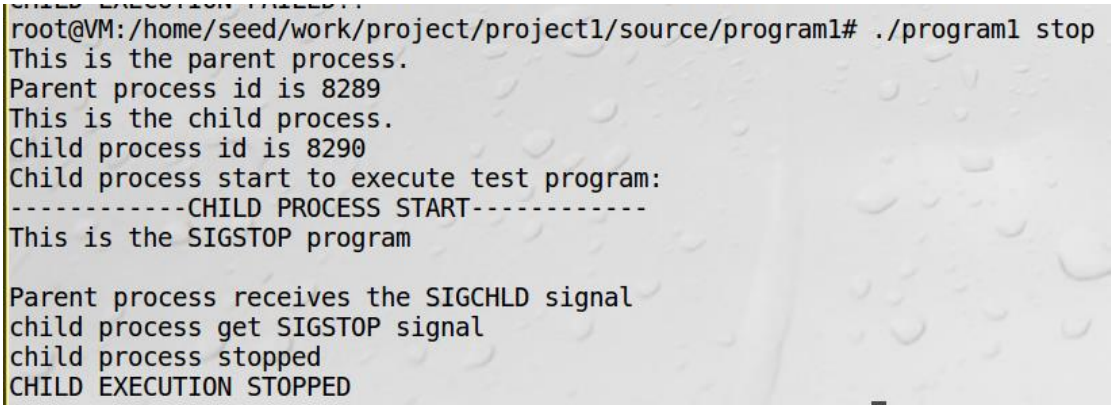
```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 pipe
This is the parent process.
Parent process id is 8260
This is the child process.
Child process id is 8261
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGPIPE program

Parent process receives the SIGCHLD signal
child process get SIGPIPE signal
child process is abort by pipe signal
CHILD EXECUTION FAILED!!
```



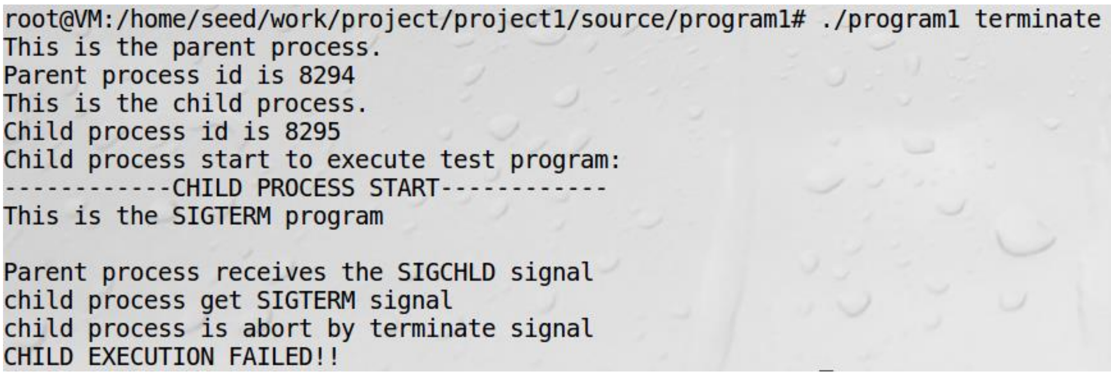
```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 quit
This is the parent process.
Parent process id is 8265
This is the child process.
Child process id is 8266
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGQUIT program

Parent process receives the SIGCHLD signal
child process get SIGQUIT signal
child process is abort by quit signal
CHILD EXECUTION FAILED!!
```



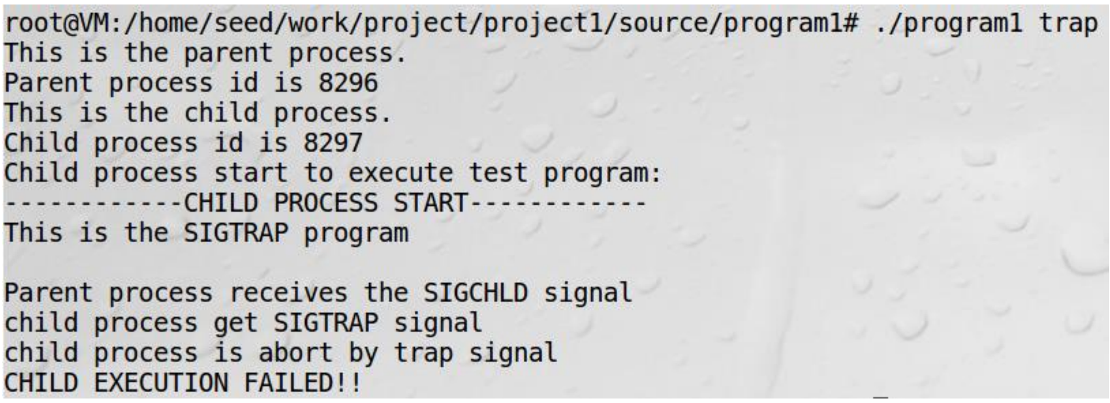
```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 segment_fault
This is the parent process.
Parent process id is 8282
This is the child process.
Child process id is 8283
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGSEGV program

Parent process receives the SIGCHLD signal
child process get SIGSEGV signal
child process is abort by segmentation fault signal
CHILD EXECUTION FAILED!!
```



```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 stop
This is the parent process.
Parent process id is 8289
This is the child process.
Child process id is 8290
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGSTOP program

Parent process receives the SIGCHLD signal
child process get SIGSTOP signal
child process stopped
CHILD EXECUTION STOPPED
```

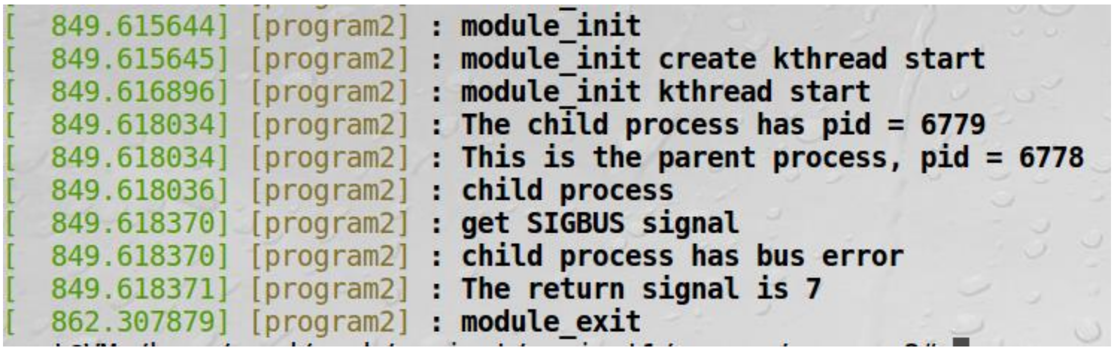


```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 terminate
This is the parent process.
Parent process id is 8294
This is the child process.
Child process id is 8295
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGTERM program

Parent process receives the SIGCHLD signal
child process get SIGTERM signal
child process is abort by terminate signal
CHILD EXECUTION FAILED!!
```

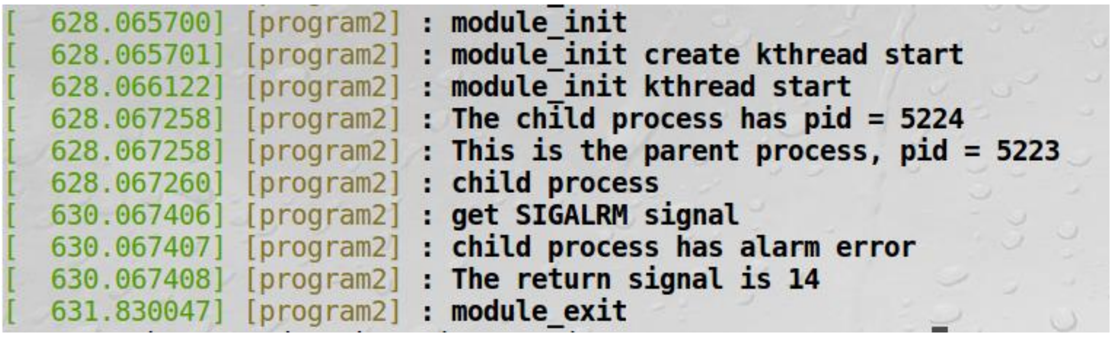


```
root@VM:/home/seed/work/project/project1/source/program1# ./program1 trap
This is the parent process.
Parent process id is 8296
This is the child process.
Child process id is 8297
Child process start to execute test program:
------------CHILD PROCESS START------------
This is the SIGTRAP program

Parent process receives the SIGCHLD signal
child process get SIGTRAP signal
child process is abort by trap signal
CHILD EXECUTION FAILED!!
```

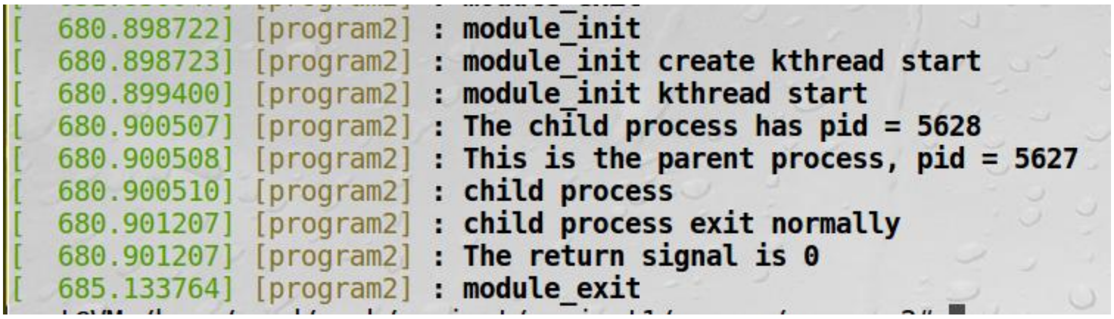
4.2. Program 2



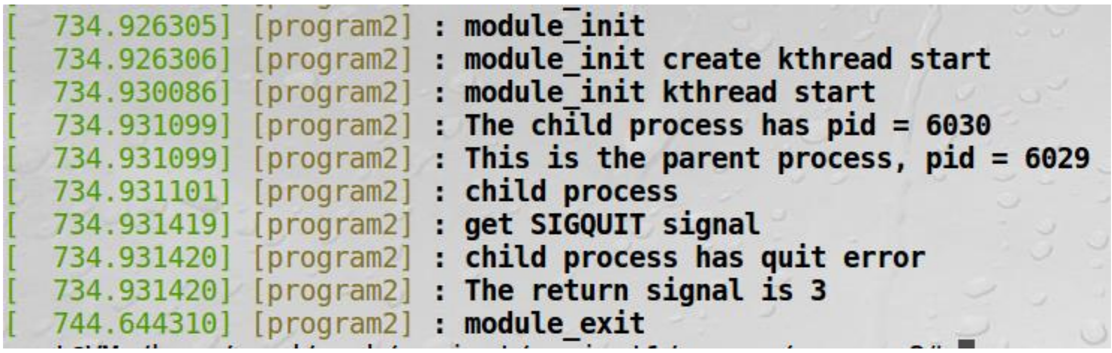
```
[  849.615644] [program2] : module_init
[  849.615645] [program2] : module_init create kthread start
[  849.616896] [program2] : module_init kthread start
[  849.618034] [program2] : The child process has pid = 6779
[  849.618034] [program2] : This is the parent process, pid = 6778
[  849.618036] [program2] : child process
[  849.618370] [program2] : get SIGBUS signal
[  849.618370] [program2] : child process has bus error
[  849.618371] [program2] : The return signal is 7
[  862.307879] [program2] : module_exit
```



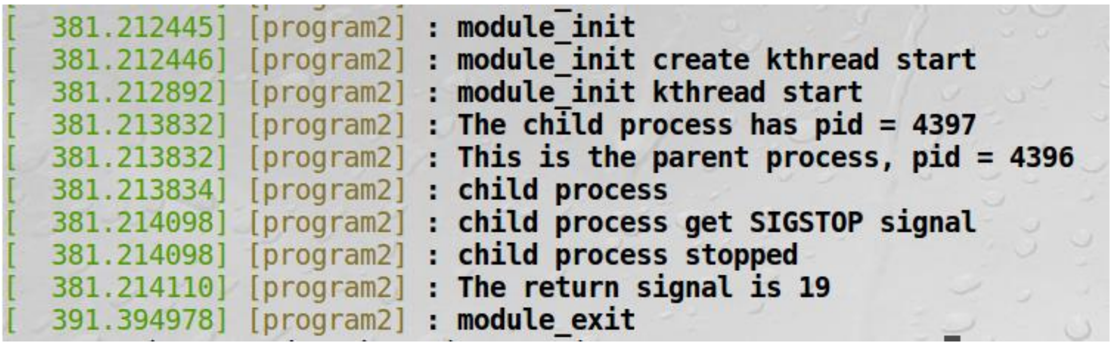
```
[  628.065700] [program2] : module_init
[  628.065701] [program2] : module_init create kthread start
[  628.066122] [program2] : module_init kthread start
[  628.067258] [program2] : The child process has pid = 5224
[  628.067258] [program2] : This is the parent process, pid = 5223
[  628.067260] [program2] : child process
[  630.067406] [program2] : get SIGALRM signal
[  630.067407] [program2] : child process has alarm error
[  630.067408] [program2] : The return signal is 14
[  631.830047] [program2] : module_exit
```



```
[  680.898722] [program2] : module_init
[  680.898723] [program2] : module_init create kthread start
[  680.899400] [program2] : module_init kthread start
[  680.900507] [program2] : The child process has pid = 5628
[  680.900508] [program2] : This is the parent process, pid = 5627
[  680.900510] [program2] : child process
[  680.901207] [program2] : child process exit normally
[  680.901207] [program2] : The return signal is 0
[  685.133764] [program2] : module_exit
```

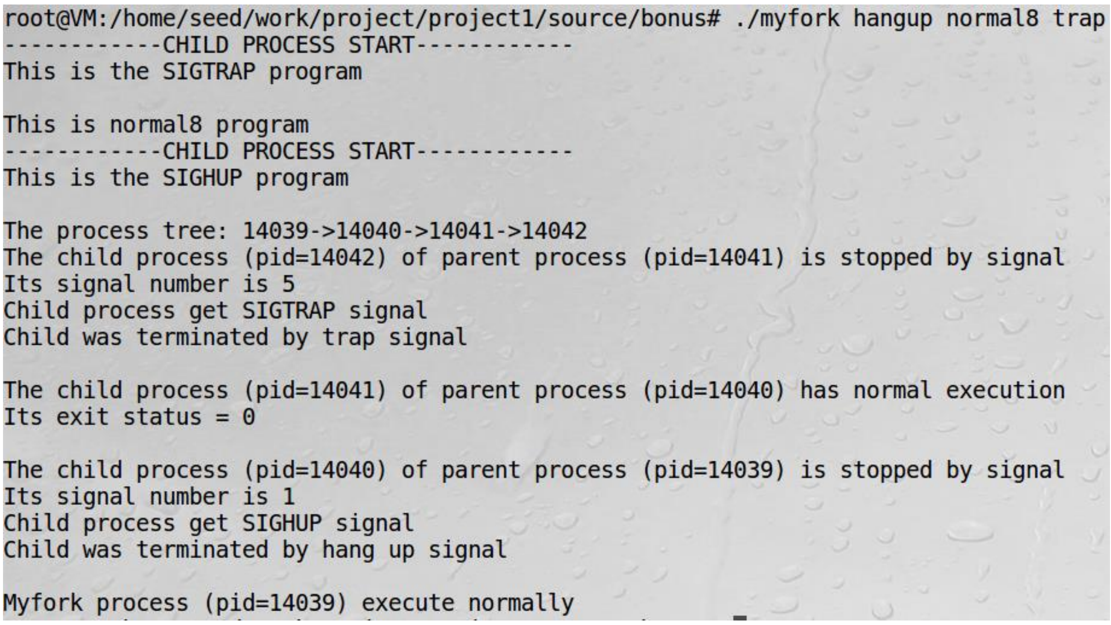


```
[  734.926305] [program2] : module_init
[  734.926306] [program2] : module_init create kthread start
[  734.930086] [program2] : module_init kthread start
[  734.931099] [program2] : The child process has pid = 6030
[  734.931099] [program2] : This is the parent process, pid = 6029
[  734.931101] [program2] : child process
[  734.931419] [program2] : get SIGQUIT signal
[  734.931420] [program2] : child process has quit error
[  734.931420] [program2] : The return signal is 3
[  744.644310] [program2] : module_exit
```



```
[  381.212445] [program2] : module_init
[  381.212446] [program2] : module_init create kthread start
[  381.212892] [program2] : module_init kthread start
[  381.213832] [program2] : The child process has pid = 4397
[  381.213832] [program2] : This is the parent process, pid = 4396
[  381.213834] [program2] : child process
[  381.214098] [program2] : child process get SIGSTOP signal
[  381.214098] [program2] : child process stopped
[  381.214110] [program2] : The return signal is 19
[  391.394978] [program2] : module_exit
```

4.3. Bonus



```
root@VM:/home/seed/work/project/project1/source/bonus# ./myfork hangup normal8 trap
------------CHILD PROCESS START------------
This is the SIGTRAP program

This is normal8 program
------------CHILD PROCESS START------------
This is the SIGHUP program

The process tree: 14039->14040->14041->14042
The child process (pid=14042) of parent process (pid=14041) is stopped by signal
Its signal number is 5
Child process get SIGTRAP signal
Child was terminated by trap signal

The child process (pid=14041) of parent process (pid=14040) has normal execution
Its exit status = 0

The child process (pid=14040) of parent process (pid=14039) is stopped by signal
Its signal number is 1
Child process get SIGHUP signal
Child was terminated by hang up signal

Myfork process (pid=14039) execute normally
```

5. What Did I Learn from The Tasks

5.1. Program 1

a. How to fork a child process

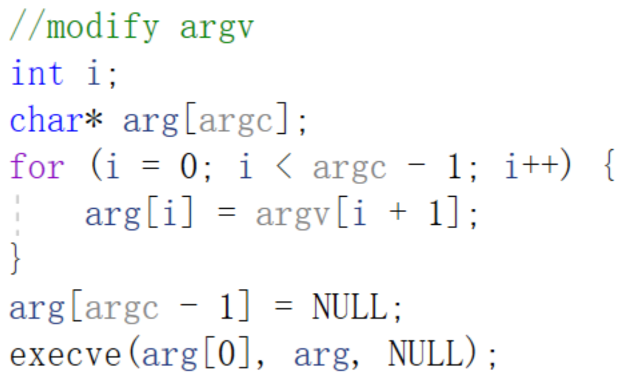
Use fork() to fork a child process. It returns the child process' process ID to the parent process and 0 to the child process. Both parent and child process will continue execution from fork(). Use if and else statements to distinguish them.

b. How to identify a fork error?

If fork() returns a negative number, there is a fork error.

c. How to execute the test program?

First, modify the original argv[]. Then call execve(arg[0], arg, NULL) to execute the test program.



```
//modify argv
int i;
char* arg[argc];
for (i = 0; i < argc - 1; i++) {
    arg[i] = argv[i + 1];
}
arg[argc - 1] = NULL;
execve(arg[0], arg, NULL);
```

d. What if the original child process continues?

There must be an error, since normally the rest of the original program should be replaced by the test program.

e. How to make the parent process wait for the child process?

Use waitpid(-1, &status, WUNTRACED). This function needs three input parameters which are pid, status and option. There are three options, 0, WNOHANG and WUNTRACED. 0 skips the option, WNOHANG requires the signal information immediately, and WUNTRACED will wait for the child process to terminate and return the signal information.

f. How to check the termination status of the child process?

For normal termination, use WIFEXITED(status) to check. For exceptions, use WIFSIGNALED(status) to check. Identify the exception signal using WTERMSIG(status). For stops, use WIFSTOPPED(status) to check and WSTOPSIG(status) to get the stop signal. If all the clauses above fail, the child process continues.

5.2. Program 2

a. What is a loadable kernel object?

A loadable kernel module (or LKM) is an object file that contains code to extend the running kernel, or so-called base kernel.

b. How to modify the Linux Kernel to export symbols?

First add EXPORT_SYMBOL() in the C source code. Then recompile and reinstall the kernel.

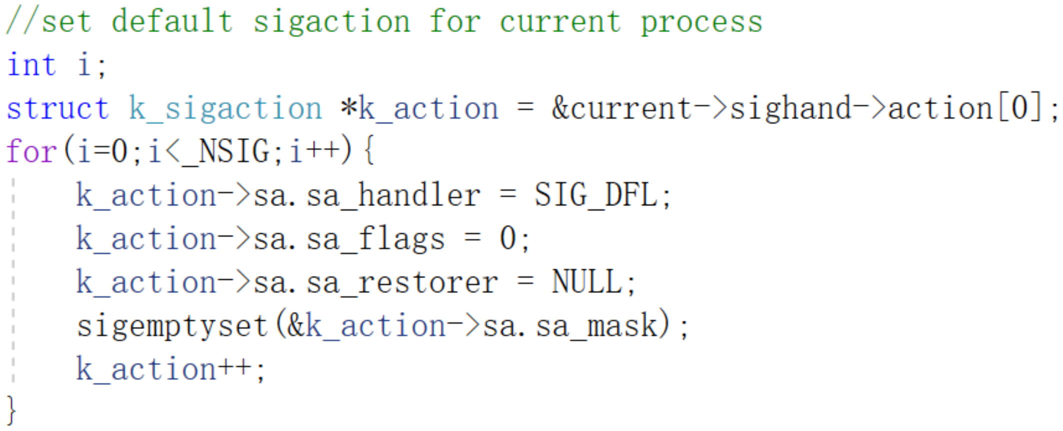
c. How to use the functions from the kernel in my program?

Add the keyword “extern” in front of the function prototype.

d. How to print information to the kernel log?

Use printk() instead of printf().

e. How to set default sigaction for the current process?



```
//set default sigaction for current process
int i;
struct k_sigaction *k_action = &current->sighand->action[0];
for(i=0;i<_NSIG;i++){
    k_action->sa.sa_handler = SIG_DFL;
    k_action->sa.sa_flags = 0;
    k_action->sa.sa_restorer = NULL;
    sigemptyset(&k_action->sa.sa_mask);
    k_action++;
}
```

f. How to fork a process in the kernel mode?

Use _do_fork() instead of fork().

g. How to execute the test program in the kernel mode?

First, we need to prepare the arguments.

```
//prepare the filename
//path[] needs to be changed when the directory is changed
const char path[] = "/home/seed/work/project/project1/source/program2/test";
const char* const argv[] = { path, NULL, NULL };
const char* const envp[] = { "HOME=/", "PATH=/sbin:/user/sbin:/bin:/usr/bin", NULL };

struct filename* my_filename = getname(path);
```

Note that path[] is an absolute path, which means it needs to be changed when the directory and/or file to execute is changed.

Then, call do_execve() function to execute the test program.

h. How to get the process ID of the current process?

Use current->pid.

i. How to make the parent process wait until the child process terminates?

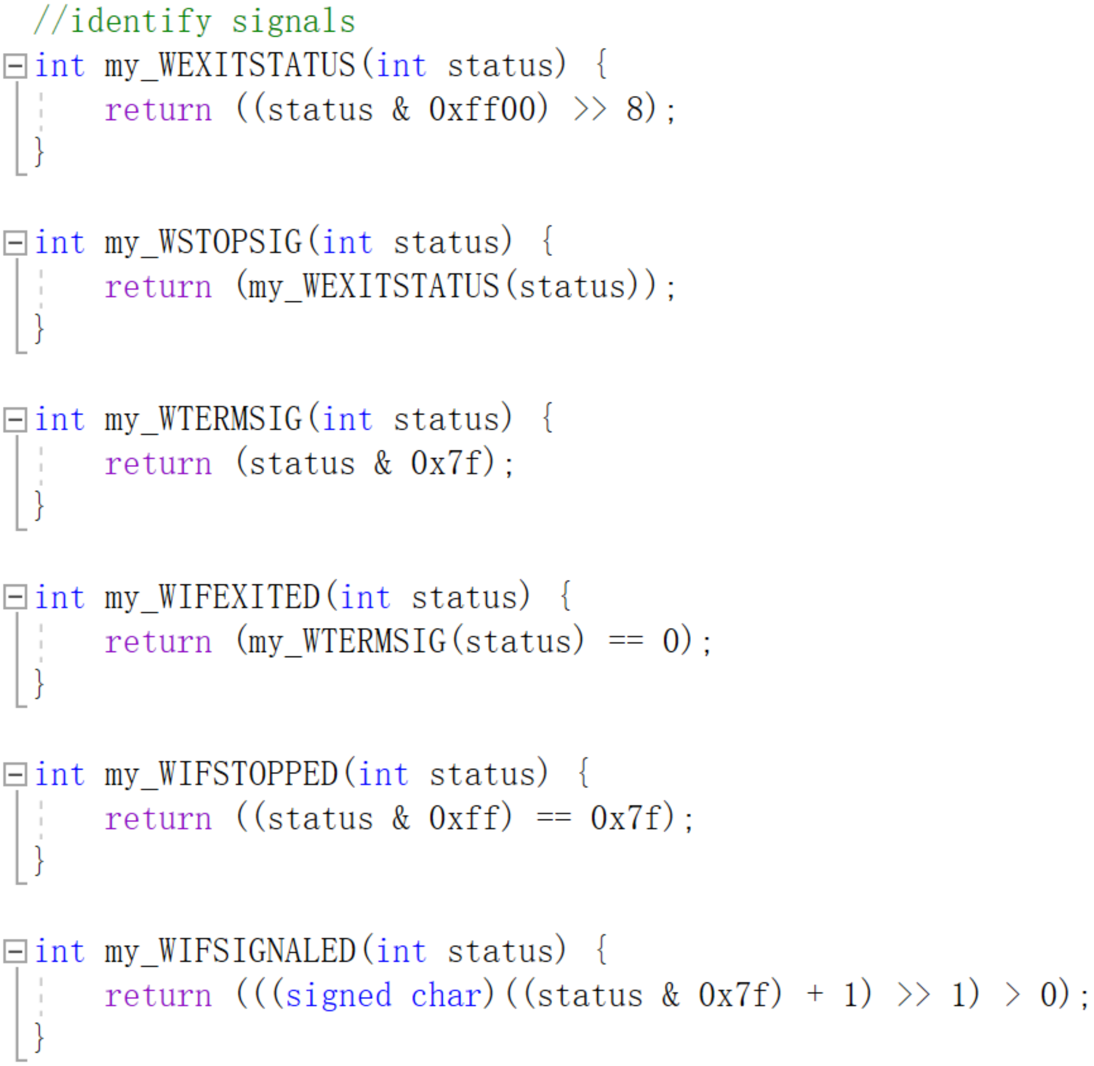
First, we need to prepare a wait_opts struct.

```
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;
```

Then, call do_wait() to make the parent process wait for the child process' termination. In the end, to decrease the count of wo_pid in the hash table and free the memory allocated, call put_pid().

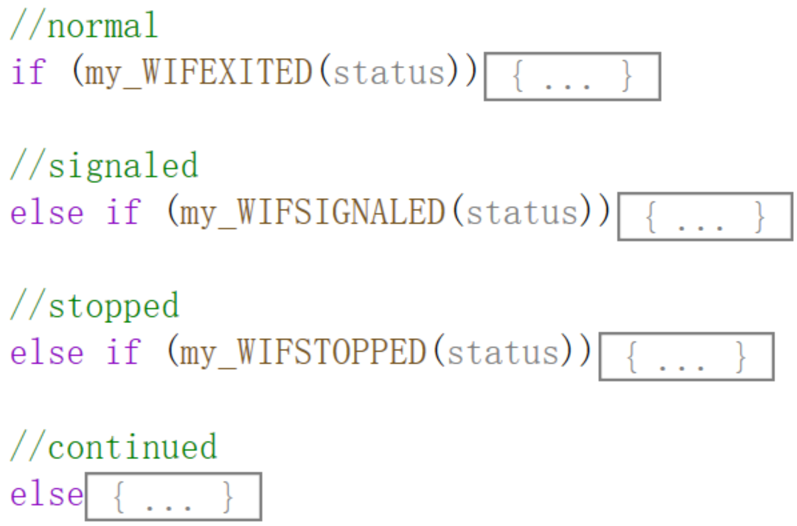
j. How to identify signals raised by the child process in kernel mode?

First, we need to implement functions to evaluate the child process' status as well as returned value of status argument by ourselves.



```
//identify signals
int my_WEXITSTATUS(int status) {
    return ((status & 0xff00) >> 8);
}

int my_WSTOPSIG(int status) {
    return (my_WEXITSTATUS(status));
}

int my_WTERMSIG(int status) {
    return (status & 0x7f);
}

int my_WIFEXITED(int status) {
    return (my_WTERMSIG(status) == 0);
}

int my_WIFSTOPPED(int status) {
    return ((status & 0xff) == 0x7f);
}

int my_WIFSIGNALED(int status) {
    return (((signed char)((status & 0x7f) + 1) >> 1) > 0);
}
```

The following steps are basically the same as those in Program 1.



```
//normal
if (my_WIFEXITED(status)) { ... }

//signaled
else if (my_WIFSIGNALED(status)) { ... }

//stopped
else if (my_WIFSTOPPED(status)) { ... }

//continued
else { ... }
```

5.3. Bonus

a. How to create a series of processes, with the latter ones being the former ones' children?

   We can fork the processes recursively. That is, we first fork the original process, and then fork the children recursively.

b. How to keep track of the process tree?

   When doing fork recursively, we can record the process IDs and termination status in an array. However, after fork(), there are multiple processes. To keep track of the process tree, they must record the information in the same place, which leads to the next question.

c. How can different processes visit the same memory location?

   We can use vfork() instead of fork() to make the processes share the same heap storage. Then, we can use a pointer to record the information.