CSC3150 HM1 Report

Ziqi MENG 120020153

Development Environment

- Linux Distribution: Ubuntu 16.04
- Linux Kernel Version: 5.10.1
- GCC Version: 5.4.0

Linux kernel preparation (5.10.x)

Step 1. Download a Linux kernel 5.10.x and get the essentials

```
root@csc3150:/home# cd /home/vagrant/csc3150
root@csc3150:/home/vagrant/csc3150# wget {url}
#{url} = https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.10.1.tar.xz
root@csc3150:/home/vagrant/csc3150# apt-get install libncurses-dev gawk\
  flex bison openssl libssl-dev dkms libelf-dev libudev-dev libpci-dev\
  libiberty-dev autoconf llvm dwarves
root@csc3150:/home/vagrant/csc3150# tar xvf linux-5.10.1.tar.xz # unpack
```

Step 2. Compile

```
cd /home
root@csc3150:/home# cd /home/seed/work
root@csc3150:/home/seed/work# mv /home/vagrant/csc3150/linux-5.10.1
/home/seed/work
root@csc3150:/home/seed/work# cd ./linux-5.15.10/
root@csc3150:/home/seed/work/linux-5.15.10# make mrproper
root@csc3150:/home/seed/work/linux-5.15.10# make clean
root@csc3150:/home/seed/work/linux-5.15.10# make menuconfig
#a menu will pop up, '->' to 'save', press 'enter', press 'enter' to save
the config, '<-' to 'exit', press 'enter'"
root@csc3150:/home/seed/work/linux-5.15.10# sudo apt-get install bc
root@csc3150:/home/seed/work/linux-5.15.10# make -j$(nproc)
root@csc3150:/home/seed/work/linux-5.15.10# make modules_install
root@csc3150:/home/seed/work/linux-5.15.10# make install
```

Export Symbol

For program2, to use certain symbols, modify the kernel source files by adding `EXPORT_SYMBOL()` after specific functions. The files and symbols are:

```
/home/seed/work/linux-5.10.1/kernel/fork.c
EXPORT_SYMBOL(kernel_clone);

/home/seed/work/linux-5.10.1/fs/exec.c
EXPORT_SYMBOL(do_execve);

/home/seed/work/linux-5.10.1/fs/namei.c
EXPORT_SYMBOL(getname);

/home/seed/work/linux-5.10.1/kernel/exit.c
EXPORT_SYMBOL(do_wait);
```

Save all the changes and compile:

```
root@csc3150:/home/seed/work/linux-5.15.10# make -j$(nproc)
root@csc3150:/home/seed/work/linux-5.15.10# make modules_install
root@csc3150:/home/seed/work/linux-5.15.10# make install
$ reboot
```

Task 1: User Mode Program

Objective

Develop a user-space program that:

- Forks a child process.
- Executes a test program in the child process.
- The parent process waits for the child to terminate.
- Handles different termination scenarios and print out.

Implementation

- Main Function and Process Forking

In the main function, fork the process with function `pid = fork()` and handle error. `pid` is the process descriptor which returns a negative number when fails, returns 0 and the child process ID when success.

```c
int main(int argc, char *argv[]){

    pid_t pid;
    int status;

    /* fork a child process */
    printf("Process start to fork\n");
    pid = fork();

    if (pid == -1){
        perror("error");
        exit(1);
    }
```

- Child Process Logic

In the child process (`pid == 0`), set up the arguments to get the file name to be executed. Get the child process id using function `getpid()` nd execute the test program using `execve()`:

```c
    if (pid == 0){
        // Child process logic
        char *arg[argc];
        for (int i = 0; i < argc - 1; i++) {
            arg[i] = argv[i + 1];
        };
        arg[argc - 1] = NULL;

        printf("I'm the Child Process, my pid = %d\n",
               getpid());


        /* execute test program */
        printf("Child process start to execute test program:\n");
        execve(arg[0], arg, NULL);
        // If execve fails, print error and terminate child process
        perror("execve");
        exit(EXIT_FAILURE);
    }
```

- Parent Process Logic

In the parent process, use function `waitpid()` to wait for the child process to terminate or stop. Handle its termination status with the help of function `get_signal_name()`:

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

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

    /* check child process'  termination status */
    // If the child terminated normally
    if (WIFEXITED(status)){
        printf("Normal termination with EXIT STATUS = %d\n", WEXITSTATUS(status));
        }
    // If the child process was stopped
    else if (WIFSTOPPED(status)){
        printf("child process get SIGSTOP signal\n");
        }
    // If the child process was terminated by a signal
    else if (WIFSIGNALED(status)){
            int sig = WTERMSIG(status);
            printf("child process get %s signal\n", get_signal_name(sig));
            }
    // Handle other cases
    else{
            printf("CHILD PROCESS CONTINUED\n");
            }

        exit(0);
}
```

- Signal Name Function

`get_signal_name(int sig_num)` maps signal numbers to their corresponding signal

names.

```
const char* get_signal_name(int sig_num) {
    switch (sig_num) {
        case 1: return "SIGHUP";
        case 2: return "SIGINT";
        case 3: return "SIGQUIT";
        case 4: return "SIGILL";
        case 5: return "SIGTRAP";
        case 6: return "SIGABRT";
        case 7: return "SIGBUS";
        case 8: return "SIGFPE";
        case 9: return "SIGKILL";
        case 11: return "SIGSEGV";
        case 13: return "SIGPIPE";
        case 14: return "SIGALRM";
        case 15: return "
        case 19: return "
        default: return "Unknown Signal";
    }
}
```

(char [15])"Unknown Signal"

Program Output

```
vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./normal
Process start to fork
I'm the Parent Process, my pid = 18994
I'm the Child Process, my pid = 18995
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
```

Normal Termination in Program 1

```
vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./abort
Process start to fork
I'm the Parent Process, my pid = 19022
I'm the Child Process, my pid = 19023
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
```

SIGABRT Signal in Program 1

```
vagrant@csc3150:~/csc3150/source/program1$ ./program1 ./stop
Process start to fork
I'm the Parent Process, my pid = 19052
I'm the Child Process, my pid = 19053
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
```

SIGSTOP Signal in Program 1

Task 2: Kernel Module

Objective

Create a Linux kernel module that:

- Creates a kernel thread.
- Forks a child process within the kernel thread using kernel_clone().
- Executes a test program in the child process using do_execve().
- Print out process id for both parent and child.
- The parent process waits for the child to terminate using do_wait().
- Handles different termination scenarios.

Implementation

1. Preparation

- Defining Macros for Status Handling: redefine macros similar to those in user space for interpreting child's termination status.

```
#define WIFEXITED(status)   (((status) & 0x7F) == 0)
#define WEXITSTATUS(status) (((status) & 0xFF00) >> 8)
// ... other macros ...
```

- Kernel thread task structure

```
static struct task_struct *task;
```

- Define the structure for do_wait parameters.

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

    struct waitid_info *wo_info;
    int wo_stat;
    struct rusage *wo_rusage;

    wait_queue_entry_t child_wait;
    int notask_error;
};
```

- Functions from kernel.

```
extern pid_t kernel_clone(struct kernel_clone_args *kargs);
extern int do_execve( struct filename *filename,
                      const char __user *const __user *__argv,
                      const char __user *const __user *__envp);
extern struct filename *getname_kernel(const char *filename);
extern long do_wait(struct wait_opts *wo);
```

2. `my_exec` Function

The `my_exec` function is responsible for executing a test program within the child process. It performs the following steps:

- Defines the path to the test program, which is /tmp/test in this case.
- Obtains a filename structure for the test program using `getname_kernel`.
- Executes the test program using `do_execve`, passing NULL for arguments and environment variables.
- Handles errors appropriately, printing error messages to the kernel log if necessary.

```
int my_exec(void)
{
    //const char path[] = "/home/vagrant/csc3150/source/program2/test";
    const char path[] = "/tmp/test";

    struct filename *file_name = getname_kernel(path);
    int result;

    if (IS_ERR(file_name)) {
        printk("[program2] : Failed to get filename, error = %ld\n", PTR_ERR(file_name));
        return PTR_ERR(file_name);
    }

    /* Execute a test program */
    result = do_execve(file_name, NULL, NULL);
    if (result < 0) {
        printk("[program2] : Failed to execute program, error = %d\n", result);
        return result;
    }

    return 0;
}
```

3. `my_wait` Function

The `my_wait` function waits for the child process to terminate. It uses the `do_wait` function to perform the wait operation. The steps involved are

- Initializes a `wait_opts` structure with appropriate parameters, such as the PID of the child process and the flags indicating the types of events to wait for (`WEXITED | WSTOPPED`).
- Calls `do_wait` to wait for the child process.
- Releases the PID reference using `put_pid`.
- Calls `handle_signal` to process the termination status of the child.

```
void my_wait(pid_t pid)
{
    int status;
    struct wait_opts wo;
    struct pid *wo_pid = NULL;
    enum pid_type type;
    type = PIDTYPE_PID;
    wo_pid = find_get_pid(pid);


    /* Wait options setup */
    wo.wo_type = type;
    wo.wo_pid=wo_pid;
    wo.wo_flags=WEXITED|WSTOPPED;
    wo.wo_info=NULL;
    wo.wo_stat=&status;
    wo.wo_rusage=NULL;

    printk("[program2] : child process\n");

    do_wait(&wo);
    put_pid(wo_pid);

    handle_signal(wo.wo_stat);
}
```

4. `handle_signal` function

The `handle_signal` function interprets the termination status of the child process and prints corresponding messages. It checks for different termination conditions:

- If the child terminated due to a signal (`WIFSIGNALED`), it retrieves the signal number and prints an appropriate message.
- If the child was stopped (`WIFSTOPPED`), it handles the stop signal.
- If the child exited normally (`WIFEXITED`), it retrieves the exit status and prints it.

```c
if (WIFSIGNALED(status)) {
    signal = WTERMSIG(status);
    printk("[program2] : get %s signal\n", signal_names[signal]);
    printk("[program2] : child process terminated\n");
    printk("[program2] : The return signal is %d\n", signal);
} else if (WIFSTOPPED(status)) {
    signal = WSTOPSIG(status);
    printk("[program2] : get SIGSTOP signal\n");
    printk("[program2] : child process stopped\n");
    printk("[program2] : the return signal is %d\n",
           signal);
} else if (WIFEXITED(status)) {
    int exit_status = WEXITSTATUS(status);
    printk("[program2] : child process exited normally\n");
    printk("[program2] : the return status is %d\n",
           exit_status);}
```

5. `my_fork` function

The my_fork function is executed by the kernel thread and performs the following operations:

- Set default sigaction for current process
- Prepares `kernel_clone_args` with the necessary flags and function pointers to create a new process.
- Calls `kernel_clone` to fork a new child process.
- Prints the `PID` of the child and parent processes.
- Calls `my_wait` to wait for the child process to terminate.

```c
int my_fork(void *argc){
    int i;
    //set default sigaction for current process
    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++;
    }

    /* fork a process using kernel_clone or kernel_thread */
    /* execute a test program in child process */
    struct kernel_clone_args clone_args
    = {
        //.flags = SIGCHLD,
        .flags = ((lower_32_bits(SIGCHLD) | CLONE_VM | CLONE_UNTRACED) & ~CSIGNAL),
        .exit_signal = SIGCHLD,
        .stack = (unsigned long)&my_exec,
        .parent_tid = NULL,
        .child_tid = NULL
    };


    pid_t pid = kernel_clone(&clone_args);
    printk("[program2] : The child process has pid = %d\n", pid);
    printk("[program2] : This is the parent process, pid = %d\n", (int)current->pid);

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

    return 0;
}
```

6. Module Initialization and Exit

- `program2_init function`

Upon module insertion, the `program2_init` function is called:

Prints an initialization message to the kernel log.

Creates a kernel thread using `kthread_create`, which runs the `my_fork` function.

Wakes up the kernel thread using `wake_up_process`.

```c
static int __init program2_init(void){

    printk("[program2] : Module_init\n");

    /* write your code here */

    /* create a kernel thread to run my_fork */
    printk("[program2] : module_init create kthread start\n");
    task = kthread_create(&my_fork, NULL, "MyThread");

    if (!IS_ERR(task))
    {
        printk("[program2] : module_init kthread start\n");
        wake_up_process(task);
    }
    return 0;
    do_exit(0);
}
```

- `program2_exit function`

Upon module removal, the `program2_exit` function is called:

Prints an exit message to the kernel log.

Program Output

```
[ 6496.120219] [program2] : Module_init
[ 6496.120221] [program2] : module_init create kthread start
[ 6496.120939] [program2] : module_init kthread start
[ 6496.121541] [program2] : The child process has pid = 2432
[ 6496.121543] [program2] : This is the parent process, pid = 2431
[ 6496.121544] [program2] : child process
[ 6496.333751] [program2] : get SIGBUS signal
[ 6496.333753] [program2] : child process terminated
[ 6496.333755] [program2] : The return signal is 7
[ 6510.593184] [program2] : Module_exit
```

What I Learned

1. Set up a virtual machine. I get more familiar with common Linux commands and knowledge about ssh.
2. Check and change the kernel version. Modifying kernel source files and recompile.
3. Insert and remove kernel modules.
4. Create and fork process in both user mode and kernel mode, and how to handle different signals.