Lab 4: Process Creation and Management

1.Objective

- Learn the creation and management of Linux process

2.Syllabus

- Understand the concepts and principle of process
- Implement the creation and management of process using C or C++

3.Prerequisite

- C or C++ language
- Computer which run Linux system (e.g. Ubuntu)
- Understand the concept of process

4.Concepts and Principles of Process

Please refer to the chapter 3 of the text book. The main contents have been lectured in the third week, 09/26/2016. The slide of this chapter can be found in the course website: http://www.thinkmesh.net/ose/.

5.Experimental Contents

5.1 Basic Program

Description: A simple program to create a child process using fork. Parent wait for child to collect the exit status of child. The function of **fork()** will create a new process. The function of **getpid()** will get the current process id. The function of **getppid()** will get the current parent process id. The function of **wait()** will wait for the child process to finish. The function of **exit()** will finish a process and exit. The file name is “**basic_fork.c**”

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<stdlib.h>

int main()
{
	int ret_val, pid, pidc;
	int ret_val2;

	/*create a child*/
	pid = fork();
	if(pid < 0) {
		perror("fork failed\n");
		exit(EXIT_FAILURE);
	}
	if(pid > 0) { /*parent code*/
```

```
        printf("parent code pid=%d\n",getpid());
        /*wait for child to terminate and catch the exit code*/
        pidc = wait(&ret_val);
        printf("pidc = %d\n",pidc);
        if(pidc == -1) {
            perror("error in wait\n");
            exit(EXIT_FAILURE);
        }

        if(WIFEXITED(ret_val)) {
            /*child terminated sucessfully*/
            printf("child exited with status %d\n",WEXITSTATUS(ret_val));
        } else if (WIFSIGNALED(ret_val))
            printf("killed by signal %d\n", WTERMSIG(ret_val));
        else if (WIFSTOPPED(ret_val))
            printf("stopped by signal %d\n", WSTOPSIG(ret_val));
        else if (WIFCONTINUED(ret_val))
            printf("continued\n");

        exit(EXIT_SUCCESS);
    } else { /*child code*/
        printf("child pid=%d, parent pid=%d\n",getpid(), getppid());
        exit(EXIT_SUCCESS);
    }
    return EXIT_SUCCESS;
}
```

How to Run: gcc -o basic_fork basic_fork.c
./basic_fork

5.2 Child-of-Child

Description: Program to make below given hierarchy using fork. The file name is "**child_of_child.c**"

```
  parent
        |
        child
             |
             child
                  |
                  child
```

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

#define NUM_CHILD 3

int main()
{
    int ret_pid;
    int ret_val;
    int i;

    for (i = 0;i < NUM_CHILD;i++) {
        ret_pid = fork();
        if (ret_pid == -1) {
            perror("error in fork:");
            exit(EXIT_FAILURE);
        }


        if (ret_pid > 0) {
            /*parent code*/
            printf("parent code, pid = %d,parent pid = %d\n",getpid(),getppid());
            break;
        } else {
            /*child code*/
            printf("child code, pid = %d,parent pid = %d\n",getpid(),getppid());
            /*last one will not have any child,so exit from here*/
            if (i == NUM_CHILD-1)
                exit(EXIT_SUCCESS);
        }
    }

    ret_pid = wait(&ret_val);
    if(ret_pid == -1) {
        perror("error in wait:");
        exit(EXIT_FAILURE);
    }
    printf("terminated pid = %d,status = %d\n",ret_pid,ret_val);
    return EXIT_SUCCESS;
}
```

How to Run: gcc -o child_of_child child_of_child.c

```
./child_of_child
```

5.3 Multiple Child

Description: Program to make 3 child process from a parent process using fork. The file name is “**multiple_child.c**”.

```
 Parent
        |
        |- child
        |
        |- child
        |
        `- child
```

```
#include<stdio.h>
```

```c
#include<stdlib.h>
#include<unistd.h>

#define NUM_CHILD 3

int main()
{
        int ret_pid;
        int ret_val;
        int i;

        for (i = 0;i < NUM_CHILD;i++) {
                ret_pid = fork();
                if (ret_pid == -1) {
                        perror("error in fork:");
                        exit(EXIT_FAILURE);
                }


                if (ret_pid > 0) {
                        /*parent code*/
                        printf("parent code, pid = %d,parent pid = %d\n",getpid(),getppid());
                } else {
                        /*child code*/
                        printf("child code, pid = %d,parent pid = %d\n",getpid(),getppid());
                        exit(EXIT_SUCCESS);
                }

        }

        for(i = 0;i < NUM_CHILD ; i++) {
                ret_pid = wait(&ret_val);
                if(ret_pid == -1) {
                        perror("error in wait:");
                        exit(EXIT_FAILURE);
                }

                printf("terminated pid = %d,status = %d\n",ret_pid,ret_val);
        }

        return EXIT_SUCCESS;
}
```

How to Run: gcc -o multiple_child multiple_child.c
./multiple_child

5.4 Open and Write File

Description: A simple program to create a child process using **fork()**. Parent opens a file and child writes some data to file. And then parent waits for the child and reads data from the same file. The file name is “**open_write.c**”

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<stdlib.h>

int main()
{
	int ret_val,pid,pidc;
	int fd;
	char buff[] = "hello to the world of linux";
	char read_buff[32];
	fd = open("test",O_CREAT|O_RDWR,S_IRUSR|S_IWUSR);
	if(fd < 0)
	{
		perror("error in oepning file");
		exit(EXIT_FAILURE);
	}

	/*create a child*/
	pid = fork();
	if(pid < 0)
	{
		perror("fork failed\n");
		exit(EXIT_FAILURE);
	}
	if(pid > 0)
	{
		/*wait for child to terminate and catch the exit code*/
		pidc = wait(&ret_val);
		if(WIFEXITED(ret_val))
		{
			printf("child exited with status %d\n",WEXITSTATUS(ret_val));
			/*parent has written in file so the see to the beginnin of file*/
			lseek(fd,0,SEEK_SET);
			ret_val = read(fd,read_buff,sizeof(read_buff));
			printf("ret_val =%d %s",ret_val,read_buff);
		}
		/*close the file and exit*/
		close(fd);
		exit(EXIT_SUCCESS);
	}
	else
	{
		/*write some data to file*/
		if(write(fd,buff,sizeof(buff) < 0)
		{
			perror("error in writing file\n");
			exit(EXIT_FAILURE);
		}
```

```
        printf("data written\n");
    }
    return EXIT_SUCCESS;
}
```

How to Run: gcc -o open_write open_write.c

```
./open_write
```

5.5 Show Process Information

Description: A basic Linux Kernel module to show the information of a process working. The file name is "**Makefile**"

```
obj-m += process_info.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	rm Module.symvers modules.order process_info.ko process_info.mod.c
process_info.mod.o process_info.o
```

The file name is "**process_info.c**"

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/sched.h>
#include <linux/rcupdate.h>
#include <linux/fdtable.h>
#include <linux/fs.h>
#include <linux/fs_struct.h>
#include <linux/dcache.h>
#include <linux/slab.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/stat.h>
#include <linux/mm.h>
#include <linux/highmem.h>
#include <asm/pgtable.h>
#define BUFSIZE 100

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Process Information");
static int pid = 0;
module_param( pid, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC( pid, "PID of the process");
static int __init processinfo_init(void) {
	printk( KERN_INFO "Starting module...\n");
	struct task_struct *task = current;
	struct task_struct *desiredTask = NULL;
```

```
    for_each_process( task) {
        if ( task->pid == pid) {
            desiredTask = task;
        }
    }

    if ( desiredTask != NULL) {
        printk( KERN_INFO "--A process is found with the PID = %d--\n", pid);
        printk( KERN_INFO "--The curently opened files information--\n");
        struct fdtable *filesTable;
        struct path fPath;
        char *filePath;
        char *buffer = (char *) kmalloc( GFP_KERNEL, BUFSIZE *
sizeof( char) );

        filesTable = files_fdtable( desiredTask->files);

        int i = 0;
        while ( filesTable->fd[i]) {
            fPath = filesTable->fd[i]->f_path;
            filePath = d_path( &fPath, buffer, BUFSIZE * sizeof( char) );
            printk( KERN_INFO "\t%s\n", filePath);
            i++;
        }

        printk( KERN_INFO "--Memory Management Information--\n" );

        struct mm_struct* mm = desiredTask->mm;
        printk( KERN_INFO "[CODE START]\t[CODE END]\t[CODE SIZE]\n");
        printk( KERN_INFO "%lx\t\t%lx\t%lu\n", mm->start_code,
            mm->end_code, mm->end_code - mm->start_code );

        printk( KERN_INFO "[DATA START]\t[DATA END]\t[DATA SIZE]\n");
        printk( KERN_INFO "%lx\t\t%lx\t%lu\n\n", mm->start_data,
            mm->end_data, mm->end_data - mm->start_data );

        printk( KERN_INFO "\nNotice: The stack data will be written in virtual
memory part.\n" );

        printk( KERN_INFO "[ARG START]\t[ARG END]\t[ARG SIZE]\n");
        printk( KERN_INFO "%lx\t\t%lx\t%lu\n\n", mm->arg_start,
            mm->arg_end, mm->arg_end - mm->arg_start );

        printk( KERN_INFO "[ENV START]\t[ENV END]\t[ENV SIZE]\n");
        printk( KERN_INFO "%lx\t\t%lx\t%lu\n\n", mm->env_start,
            mm->env_end, mm->env_end - mm->env_start );

        printk( KERN_INFO "Total VM area = %lu\n", mm->total_vm);
        printk( KERN_INFO "Number of frames used by the process = %lu\n\n",
get_mm_rss( mm) );
```

```
        struct vm_area_struct *mmap = mm->mmap;

        printk( KERN_INFO "--Virtual Memory Information--\n" );
        printk( KERN_INFO "[VM START]\t[VM_END]\t[VM_SIZE]");
        while( mmap != NULL )
        {
            if( mmap -> vm_next == NULL ) {
                printk( KERN_INFO "\nThe stack information of the
process:\n");
                printk( KERN_INFO "[STACK START]\t[STACK
END]\t[STACK SIZE]\n" );
            }
            printk( KERN_INFO "%lx\t\t%lx\t%lu\n", mmap -> vm_start,
                mmap -> vm_end, mmap -> vm_end - mmap -> vm_start );
            mmap = mmap -> vm_next;
        }

        printk( KERN_INFO "--The filesystem information--\n");
        struct fs_struct *filesStruct = desiredTask->fs;
        printk( KERN_INFO "\tRoot: %s\n",
filesStruct->root.dentry->d_name.name);
        printk( KERN_INFO "\tWorking Directory: %s\n",
filesStruct->pwd.dentry->d_name.name);
    }
    else {
        printk( KERN_INFO "There is not a process with PID = %d, exiting...\n",
pid);
    }

    return 0;
}

static void __exit processinfo_exit( void) {
    printk( KERN_INFO "The module successfully removed.\n");
}

module_init( processinfo_init);
module_exit( processinfo_exit)
```

How to Run: make

```
              sudo insmod process_info.ko pid=1    //install Linux kernel module
                                                   //pid can be changed to other id
              dmesg //show the detailed message output by process_info.ko
```

6.Conclusion

In this chapter, five experiments have been complete. They have shown how to create and manage a process.