Project 1 —UNIX Shell

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. Your implementation will support input and output redirection, as well as pipes as a form of IPC between a pair of commands. Completing this project will involve using the UNIX fork(), exec(), wait(), dup2(), and pipe() system calls and can be completed on any Linux, UNIX, or macOS system.

I. Overview

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt osh> and the user's next command: cat prog.c. (This command displays the file prog.c on the terminal using the UNIX cat command.)

osh>cat prog.c

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, cat prog.c) and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to the new process creation illustrated in Figure 1. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (&) at the end of the command. Thus, if we rewrite the above command as

osh>cat prog.c &

the parent and child processes will run concurrently.

The separate child process is created using the fork() system call, and the user's command is executed using one of the system calls in the exec() family.

A C program that provides the general operations of a command-line shell is supplied in the code snippet of Figure 1 below. The main() function presents the prompt osh-> and outlines the steps to be taken after input from the user has been read. The main() function continually loops as long as should_run equals 1; when the user enters exit at the prompt, your program will set should_run to 0 and terminate.

This project is organized into several parts:

1. Creating the child process and executing the command in the child

2. Providing a history feature

3. Adding support of input and output redirection

4. Allowing the parent and child processes to communicate via a pipe

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

#define MAX_LINE 80 /* The maximum length command */

int main(void)
{
char *args[MAX_LINE/2 + 1]; /* command line arguments */
int should_run = 1; /* flag to determine when to exit program */

   while (should_run) {
      printf("osh>");
      fflush(stdout);

      /**
       * After reading user input, the steps are:
       * (1) fork a child process using fork()
       * (2) the child process will invoke execvp()
       * (3) parent will invoke wait() unless command included &
       */
   }

    return 0;
}
```

Figure 1: Outline of simple shell

II. Executing Command in a Child Process

The first task is to modify the main() function in Figure 1 so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings (args in Figure 1). For example, if the user enters the command ps -ael at the osh> prompt, the values stored in the args array are:

args[0] = "ps"

args[1] = "-ael"

args[2] = NULL

This args array will be passed to the execvp() function, which has the following prototype:

execvp(char *command, char *params[])

Here, command represents the command to be performed and params stores the parameters to this command. For this project, the execvp() function should be invoked as execvp(args[0], args). Be sure to check whether the user included & to determine whether or not the parent process is to wait for the child to exit.

III. Creating a History Feature

The next task is to modify the shell interface program so that it provides a history feature to allow a user to execute the most recent command by entering ! !. For example, if a user enters the command ls -l, she can then execute that command again by entering ! ! at the prompt. Any command executed in this fashion should be echoed on the user’s screen, and the command should also be placed in the history buffer as the next command.

Your program should also manage basic error handling. If there is no recent command in the history, entering ! ! should result in a message “No commands in history.

IV. Redirecting Input and Output

Your shell should then be modified to support the ‘>’ and ‘<’ redirection operators, where ‘>’ redirects the output of a command to a file and ‘<’ redirects the input to a command from a file. For example, if a user enters

osh>ls > out.txt

the output from the ls command will be redirected to the fileout.txt. Similarly, input can be redirected as well. For example, if the user enters

osh>sort < in.txt

the file in.txt will serve as input to the sort command.

Managing the redirection of both input and output will involve using the dup2() function, which duplicates an existing file descriptor to another file descriptor. For example, if fd is a file descriptor to the file out.txt, the call

dup2(fd, STDOUT_FILENO);

duplicates fd to standard output (the terminal). This means that any writes to standard output will in fact be sent to the out.txt file.

V. Communication via a Pipe

The final modification to your shell is to allow the output of one command to serve as input to another using a pipe. For example, the following command sequence

osh>ls -l | less

has the output of the command ls -l serve as the input to the less command. Both the ls and less commands will run as separate processes and will communicate using the UNIX pipe() function.

Perhaps the easiest way to create these separate processes is to have the parent process create the child process (which will execute ls -l). This child will also create another child process (which will execute less) and will establish a pipe between itself and the child process it creates. Implementing pipe functionality will also require using the dup2() function as described in the previous section.

Finally, although several commands can be chained together using multiple pipes, you can assume that commands will contain only one pipe character and will not be combined with any redirection operators.

Project 2: Introduction to Linux Kernel Modules

In this project, you will learn how to create a kernel module and load it into the Linux kernel. You will then modify the kernel module so that it creates an entry in the /proc file system. The project can be completed using a Linux virtual machine. Although you may use any text editor to write these C programs, you will have to use the terminal application to compile the programs, and you will have to enter commands on the command line to manage the modules in the kernel.

As you'll discover, the advantage of developing kernel modules is that it is a relatively easy method of interacting with the kernel, thus allowing you to write programs that directly invoke kernel functions. It is important for you to keep in mind that you are indeed writing kernel code that directly interacts with the kernel. That normally means that any errors in the code could crash the system! However, since you will be using a virtual machine, any failures will at worst only require rebooting the system.

I. Kernel Modules Overview

The first part of this project involves following a series of steps for creating and inserting a module into the Linux kernel.

You can list all kernel modules that are currently loaded by entering the command

lsmod

This command will list the current kernel modules in three columns: name, size, and where the module is being used.

The program in Figure 2 bellow illustrates a very basic kernel module that prints appropriate messages when it is loaded and unloaded.

The function simple_init() is the module entry point, which represents the function that is invoked when the module is loaded into the kernel. Similarly, the simple_exit() function is the module exit point—the function that is called when the module is removed from the kernel.

The module entry point function must return an integer value, with 0 representing success and any other value representing failure. The module exit point function returns void. Neither the module entry point nor the module exit point is passed any parameters. The two following macros are used for registering the module entry and exit points with the kernel:

module_init(simple_init)

module_exit(simple_exit)

Notice in the figure how the module entry and exit point functions make calls to the printk() function. printk() is the kernel equivalent of printf(), but its output is sent to a kernel log buffer whose contents can be read by the dmesg command. One difference between printf() and printk() is that printk()allows us to specify a priority flag, whose values are given in the <linux/printk.h>include file. In this instance, the priority is KERNINFO, which is defined as an informational message.

The final lines—MODULE_LICENSE(), MODULE_DESCRIPTION(), and MODULE_AUTHOR()—represent details regarding the software license, description of the module, and author. For our purposes, we do not require this information, but we include it because it is standard practice in developing kernel modules.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

/* This function is called when the module is loaded. */
int simple_init(void)
{
  printk(KERN_INFO "Loading Kernel Module\n");

  return 0;
}

/* This function is called when the module is removed. */
void simple_exit(void)
{
  printk(KERN_INFO "Removing Kernel Module\n");
}

/* Macros for registering module entry and exit points. */
module_init(simple_init);
module_exit(simple_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");
```

Figure 2: Kernel module simple.c.

This kernel module simple.c is compiled using the Makefile accompanying the source code with this project. To compile the module, enter the following on the command line:

make

The compilation produces several files. The file simple.ko represents the compiled kernel module. The following step illustrates inserting this module into the Linux kernel.

II. Loading and Removing Kernel Modules

Kernel modules are loaded using the insmod command, which is run as follows:

sudo insmod simple.ko

To check whether the module has loaded, enter the lsmod command and search for the module simple. Recall that the module entry point is invoked when the module is inserted into the kernel. To check the contents of this message in the kernel log buffer, enter the command

dmesg

You should see the message "Loading Module."

Removing the kernel module involves invoking the rmmod command (notice that the .ko suffix is unnecessary):

sudo rmmod simple

Be sure to check with the dmesg command to ensure the module has been removed.

Because the kernel log buffer can fill up quickly, it often makes sense to clear the buffer periodically. This can be accomplished as follows:

sudo dmesg -c

Proceed through the steps described above to create the kernel module and to load and unload the module. Be sure to check the contents of the kernel log buffer using dmesg to ensure that you have followed the steps properly.

As kernel modules are running within the kernel, it is possible to obtain values and call functions that are available only in the kernel and not to regular user applications. For example, the Linux include file <linux/hash.h> defines several hashing functions for use within the kernel. This file also defines the constant value GOLDEN_RATIO_PRIME (which is defined as an unsigned long).

This value can be printed out as follows:

printk(KERNINFO "%lu\n", GOLDENRATIOPRIME);

As another example, the include file <linux/gcd.h> defines the following function

unsigned long gcd(unsigned long a, unsigned b);

which returns the greatest common divisor of the parameters a and b.

Once you are able to correctly load and unload your module, complete the following additional steps:

1. Print out the value of GOLDEN_RATIO_PRIME in the simple_init() function.

2. Print out the greatest common divisor of 3,300 and 24 in the simple_exit() function.

As compiler errors are not often helpful when performing kernel development, it is important to compile your program often by running makeregularly. Be sure to load and remove the kernel module and check the kernel log buffer using dmesg to ensure that your changes to simple.c are working properly.

In Chapter 1 of the course, we described the role of the timer as well as the timer interrupt handler. In Linux, the rate at which the timer ticks (the tick rate) is the value HZ defined in <asm/param.h>. The value of HZ determines the frequency of the timer interrupt, and its value varies by machine type and architecture.

For example, if the value of HZ is 100, a timer interrupt occurs 100 times per second, or every 10 milliseconds. Additionally, the kernel keeps track of the global variable jiffies, which maintains the number of timer interrupts that have occurred since the system was booted. The jiffies variable is declared in the file<linux/jiffies.h>.

1. Print out the values of jiffies and HZ in the simple_init() function.

2. Print out the value of jiffies in the simple_exit() function.

Before proceeding to the next set of exercises, consider how you can use the different values of jiffies in simple_init() and simple_exit() to determine the number of seconds that have elapsed since the time the kernel module was loaded and then removed.

III. The /proc File System

The /proc file system is a “pseudo” file system that exists only in kernel memory and is used primarily for querying various kernel and per-process statistics.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "hello"

ssize_t proc_read(struct file *file, char __user *usr_buf,
   size_t count, loff_t *pos);

static struct file_operations proc_ops = {
    .owner = THIS_MODULE,
    .read = proc_read,
};

/* This function is called when the module is loaded. */
int proc_init(void)
{
   /* creates the /proc/hello entry */
   proc_create(PROC_NAME, 0666, NULL, &proc_ops);

   return 0;
}

/* This function is called when the module is removed. */
void proc_exit(void)
{
   /* removes the /proc/hello entry */
   remove_proc_entry(PROC_NAME, NULL);
}
```

Figure 3: The /proc file-system kernel module, Part 1

```c
/* This function is called each time /proc/hello is read */
ssize_t proc_read(struct file *file, char __user *usr_buf,
   size_t count, loff_t *pos)
{
   int rv = 0;
   char buffer[BUFFER_SIZE];
   static int completed = 0;

   if (completed) {
    completed = 0;
    return 0;
   }

   completed = 1;

   rv = sprintf(buffer, "Hello World\n");

   /* copies kernel space buffer to user space usr_buf */
   copy_to_user(usr_buf, buffer, rv);

   return rv;
}
module_init(proc_init);
module_exit(proc_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Hello Module");
MODULE_AUTHOR("SGG");
```

Figure 4: The /proc file system kernel module, Part 2

This exercise involves designing kernel modules that create additional entries in the /proc file system involving both kernel statistics and information related to specific processes. The entire program is included in Figure 3 and Figure 4.

We begin by describing how to create a new entry in the/proc file system. The following program example (named hello.c found in Project 2 folder) creates a/proc entry named/proc/hello. If a user enters the command

cat /proc/hello

the infamous Hello World message is returned.

In the module entry point proc_init(), we create the new /proc/hello entry using the proc_create() function. This function is passed proc_ops, which contains a reference to a struct file_operations. This struct initializes the .owner and .read members. The value of .read is the name of the function proc_read() that is to be called whenever /proc/hello is read.

Examining this proc_read() function, we see that the string "Hello World\n" is written to the variable buffer where buffer exists in kernel memory. Since /proc/hello can be accessed from user space, we must copy the contents of buffer to user space using the kernel function copy_to_user().

This function copies the contents of kernel memory buffer to the variable usr_buf, which exists in user space.

Each time the/proc/hello file is read, the proc_read() function is called repeatedly until it returns 0, so there must be logic to ensure that this function returns 0 once it has collected the data (in this case, the string "Hello World\n") that is to go into the corresponding /proc/hello file.

Finally, notice that the/proc/hello file is removed in the module exit point proc_exit() using the function remove_proc_entry().

IV. Assignment

This assignment will involve designing two kernel modules:

1. Design a kernel module that creates a /proc file named /proc/jiffies that reports the current value of jiffies when the /proc/jiffies file is read, such as with the command

   cat /proc/jiffies

   Be sure to remove/proc/jiffies when the module is removed.
2. Design a kernel module that creates a proc file named /proc/seconds that reports the number of elapsed seconds since the kernel module was loaded. This will involve using the value of jiffies as well as the HZ rate. When a user enters the command

   cat /proc/seconds

   your kernel module will report the number of seconds that have elapsed since the kernel module was first loaded. Be sure to remove /proc/seconds when the module is removed.

Project 3: Scheduling Algorithms

This project involves implementing several different process scheduling algorithms. The scheduler will be assigned a predefined set of tasks and will schedule the tasks based on the selected scheduling algorithm. Each task is assigned a priority and CPU burst. The following scheduling algorithms will be implemented:

- First-come, first-served (FCFS), which schedules tasks in the order in which they request the CPU.
- Shortest-job-first (SJF), which schedules tasks in order of the length of the tasks' next CPU burst.
- Priority scheduling, which schedules tasks based on priority.
- Round-robin (RR) scheduling, where each task is run for a time quantum (or for the remainder of its CPU burst).
- Priority with round-robin, which schedules tasks in order of priority and uses round-robin scheduling for tasks with equal priority.

Priorities range from 1 to 10, where a higher numeric value indicates a higher relative priority. For round-robin scheduling, the length of a time quantum is 10 milliseconds.

I. Implementation

The implementation of this project may be completed in either C or Java, and program files supporting the language are provided in project 3 folder. These supporting files read in the schedule of tasks, insert the tasks into a list, and invoke the scheduler.

The schedule of tasks has the form [task name] [priority] [CPU burst], with the following example format:

T1, 4, 20

T2, 2, 25

T3, 3, 25

T4, 3, 15

T5, 10, 10

Thus, task T1 has priority 4 and a CPU burst of 20 milliseconds, and so forth. It is assumed that all tasks arrive at the same time, so your scheduler algorithms do not have to support higher-priority processes pre-empting processes with lower priorities. In addition, tasks do not have to be placed into a queue or list in any particular order.

There are a few different strategies for organizing the list of tasks, as first presented in Chapter 5. One approach is to place all tasks in a single unordered list, where the strategy for task selection depends on the scheduling algorithm. For example, SJF scheduling would search the list to find the task with the shortest next CPU burst. Alternatively, a list could be ordered according to scheduling criteria (that is, by priority). One other strategy involves having a separate queue for each unique priority, as shown in Figure 5 below. These approaches are briefly discussed in Chapter 5. It is also worth highlighting that we are using the terms list and queue somewhat interchangeably.

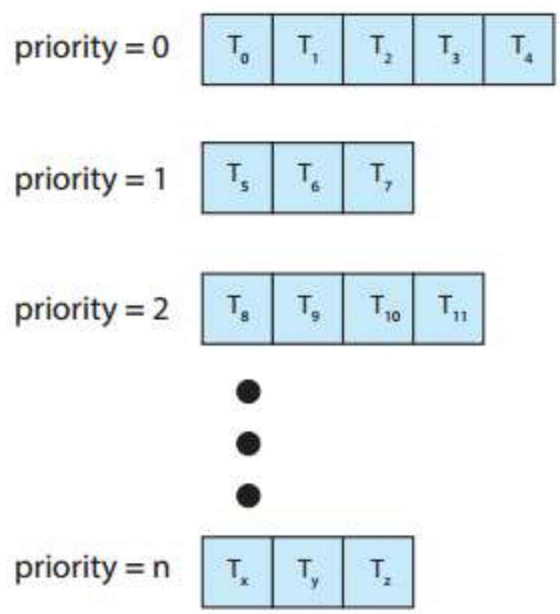

Figure 5: Separate queues for each priority.

However, a queue has very specific FIFO functionality, whereas a list does not have such strict insertion and deletion requirements. You are likely to find the functionality of a general list to be more suitable when completing this project.

II. C Implementation Details

The file driver.c reads in the schedule of tasks, inserts each task into a linked list, and invokes the process scheduler by calling the schedule() function. The schedule() function executes each task according to the specified scheduling algorithm. Tasks selected for execution on the CPU are determined by the pickNextTask() function and are executed by invoking the run()function defined in the CPU.c file. A Makefile is used to determine the specific scheduling algorithm that will be invoked by driver. For example, to build the FCFS scheduler, we would enter

make fcfs

and would execute the scheduler (using the schedule of tasksschedule.txt) as follows:

./fcfs schedule.txt

Refer to the README file in the source code download for further details. Before proceeding, be sure to familiarize yourself with the source code provided as well as the Makefile.

III. Further Challenge

Calculate the average turnaround time, waiting time, and response time for each of the scheduling algorithms.