Experiment 1

Deadline: 2024/3/31 23:59

- **Programming Projects 1**
  Textbook (Version 9) P96

Task 1: (Linux Kernel Modules) Proceed through the steps described in the textbook 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 you have properly followed the steps.

Task 2: (Kernel Data Structures) Modify the module constructed by Task 1, and complete the following two sub-tasks.
(1) In the module entry point, create a linked list containing five struct birthday elements. Traverse the linked list and output its contents to the kernel log buffer. Invoke the dmesg command to ensure the list is properly constructed once the kernel module has been loaded.
(2) In the module exit point, delete the elements from the linked list and return the free memory back to the kernel. Traverse the linked list again to ensure that the list has been removed. Finally, invoke the dmesg command to check that the list has been removed once the kernel module has been unloaded.

- **Programming Project 2**
  Textbook (version 9) P157 / P159

Task: Choose one from Project 1 and 2.
  Project 1—UNIX Shell and History Feature
  Project 2—Linux Kernel Module for Listing Tasks

Notes:
(1) Please submit all your source codes.
(2) Please submit your report, showing your experimental procedures and results. Anything else that you would like to report correlated to this project is also welcomed.
(3) You may refer to any other materials besides the textbook for completing the tasks, and please cite them properly in your report.

Programming Projects

Linux Kernel Modules

In this project, you will learn how to create a kernel module and load it into the Linux kernel. The project can be completed using the Linux virtual machine that is available with this text. Although you may use an 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.

Part I—Creating Kernel Modules

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 following program (named `simple.c` and available with the source code for this text) illustrates a very basic kernel module that prints appropriate messages when the kernel module is loaded and unloaded.

```
#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 Module\n");

  return 0;
}

/* This function is called when the module is removed. */
void simple_exit(void)
{
  printk(KERN_INFO "Removing 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");
```

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()

module_exit()
```

Notice how both the module entry and exit point functions make calls to the `printk()` function. `printk()` is the kernel equivalent of `printf()`, yet 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 `KERN_INFO`, 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 depend on this information, but we include it because it is standard practice in developing kernel modules.

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.

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
```

Part I Assignment

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 you have properly followed the steps.

Part II—Kernel Data Structures

The second part of this project involves modifying the kernel module so that it uses the kernel linked-list data structure.

In Section 1.10, we covered various data structures that are common in operating systems. The Linux kernel provides several of these structures. Here, we explore using the circular, doubly linked list that is available to kernel developers. Much of what we discuss is available in the Linux source code—in this instance, the include file `<linux/list.h>`—and we recommend that you examine this file as you proceed through the following steps.

Initially, you must define a `struct` containing the elements that are to be inserted in the linked list. The following C `struct` defines birthdays:

```
struct birthday {
  int day;
  int month;
  int year;
  struct list_head list;
}
```

Notice the member `struct list_head list`. The `list_head` structure is defined in the include file `<linux/types.h>`. Its intention is to embed the linked list within the nodes that comprise the list. This `list_head` structure is quite simple—it merely holds two members, `next` and `prev`, that point to the next and previous entries in the list. By embedding the linked list within the structure, Linux makes it possible to manage the data structure with a series of ***macro*** functions.

Inserting Elements into the Linked List

We can declare a `list_head` object, which we use as a reference to the head of the list by using the `LIST_HEAD()` macro

```
static LIST_HEAD(birthday_list);
```

This macro defines and initializes the variable `birthday_list`, which is of type `struct list_head`.

We create and initialize instances of `struct birthday` as follows:

```
struct birthday *person;

person = kmalloc(sizeof(*person), GFP_KERNEL);
person->day = 2;
person->month= 8;
person->year = 1995;
INIT_LIST_HEAD(&person->list);
```

The `kmalloc()` function is the kernel equivalent of the user-level `malloc()` function for allocating memory, except that kernel memory is being allocated. (The *GFP_KERNEL* flag indicates routine kernel memory allocation.) The macro `INIT_LIST_HEAD()` initializes the `list` member in `struct birthday`. We can then add this instance to the end of the linked list using the `list_add_tail()` macro:

```
list_add_tail(&person->list, &birthday_list);
```

Traversing the Linked List

Traversing the list involves using the `list_for_each_entry()` Macro, which accepts three parameters:

- A pointer to the structure being iterated over
- A pointer to the head of the list being iterated over
- The name of the variable containing the `list_head` structure

The following code illustrates this macro:

```
struct birthday *ptr;

list_for_each_entry(ptr, &birthday_list, list) {
  /* on each iteration ptr points */
  /* to the next birthday struct */
}
```

Removing Elements from the Linked List

Removing elements from the list involves using the `list_del()` macro, which is passed a pointer to `struct list_head`

```
list_del(struct list_head *element)
```

This removes *element* from the list while maintaining the structure of the remainder of the list.

Perhaps the simplest approach for removing all elements from a linked list is to remove each element as you traverse the list. The macro `list_for_each_entry_safe()` behaves much like `list_for_each_entry()`

except that it is passed an additional argument that maintains the value of the `next` pointer of the item being deleted. (This is necessary for preserving the structure of the list.) The following code example illustrates this macro:

```
struct birthday *ptr, *next

list_for_each_entry_safe(ptr,next,&birthday_list,list) {
   /* on each iteration ptr points */
   /* to the next birthday struct */
   list_del(&ptr->list);
   kfree(ptr);
}
```

Notice that after deleting each element, we return memory that was previously allocated with `kmalloc()` back to the kernel with the call to `kfree()`. Careful memory management—which includes releasing memory to prevent ***memory leaks***—is crucial when developing kernel-level code.

Part II Assignment

In the module entry point, create a linked list containing five `struct birthday` elements. Traverse the linked list and output its contents to the kernel log buffer. Invoke the `dmesg` command to ensure the list is properly constructed once the kernel module has been loaded.

In the module exit point, delete the elements from the linked list and return the free memory back to the kernel. Again, invoke the `dmesg` command to check that the list has been removed once the kernel module has been unloaded.

Programming Projects

Project 1—UNIX Shell and History Feature

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. This project can be completed on any Linux, UNIX, or Mac OS X system.

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 3.10. 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 (as described in Section 3.3.1).

A C program that provides the general operations of a command-line shell is supplied in Figure 3.36. 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 two parts: (1) creating the child process and executing the command in the child, and (2) modifying the shell to allow a history feature.

```
#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) if command included &, parent will invoke wait()
       */
   }

    return 0;
}
```

Figure 3.36 Outline of simple shell.

Part I— Creating a Child Process

The first task is to modify the `main()` function in Figure 3.36 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 3.36). 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 an & to determine whether or not the parent process is to wait for the child to exit.

Part II—Creating a History Feature

The next task is to modify the shell interface program so that it provides a ***history*** feature that allows the user to access the most recently entered commands. The user will be able to access up to 10 commands by using the feature. The commands will be consecutively numbered starting at 1, and the numbering will continue past 10. For example, if the user has entered 35 commands, the 10 most recent commands will be numbered 26 to 35.

The user will be able to list the command history by entering the command

```
history
```

at the `osh>` prompt. As an example, assume that the history consists of the commands (from most to least recent):

```
ps, ls -l, top, cal, who, date
```

The command `history` will output:

```
6 ps
5 ls -l
4 top
3 cal
2 who
1 date
```

Your program should support two techniques for retrieving commands from the command history:

1. When the user enters `!!`, the most recent command in the history is executed.
2. When the user enters a single `!` followed by an integer N, the N^{th} command in the history is executed.

Continuing our example from above, if the user enters `!!`, the ps command will be performed; if the user enters `!3`, the command `cal` will be executed. Any command executed in this fashion should be echoed on the user's screen. The command should also be placed in the history buffer as the next command.

The program should also manage basic error handling. If there are no commands in the history, entering `!!` should result in a message "`No commands in history.`" If there is no command corresponding to the number entered with the single `!`, the program should output "`No such command in history.`"

Project 2—Linux Kernel Module for Listing Tasks

In this project, you will write a kernel module that lists all current tasks in a Linux system. Be sure to review the programming project in Chapter 2, which deals with creating Linux kernel modules, before you begin this project. The project can be completed using the Linux virtual machine provided with this text.

Part I—Iterating over Tasks Linearly

As illustrated in Section 3.1, the PCB in Linux is represented by the structure `task_struct`, which is found in the `<linux/sched.h>` include file. In Linux, the `for_each_process()` macro easily allows iteration over all current tasks in the system:

```
#include <linux/sched.h>

struct task_struct *task;

for_each_process(task) {
  /* on each iteration task points to the next task */
}
```

The various fields in `task_struct` can then be displayed as the program loops through the `for_each_process()` macro.

Part I Assignment

Design a kernel module that iterates through all tasks in the system using the `for_each_process()` macro. In particular, output the task name (known as ***executable name***), state, and process id of each task. (You will probably have to read through the `task_struct` structure in `<linux/sched.h>` to obtain the names of these fields.) Write this code in the module entry point so that its contents will appear in the kernel log buffer, which can be viewed using the `dmesg` command. To verify that your code is working correctly, compare the contents of the kernel log buffer with the output of the following command, which lists all tasks in the system:

```
ps -el
```

The two values should be very similar. Because tasks are dynamic, however, it is possible that a few tasks may appear in one listing but not the other.

Part II—Iterating over Tasks with a Depth-First Search Tree

The second portion of this project involves iterating over all tasks in the system using a depth-first search (DFS) tree. (As an example: the DFS iteration of the processes in Figure 3.8 is 1, 8415, 8416, 9298, 9204, 2, 6, 200, 3028, 3610, 4005.)

Linux maintains its process tree as a series of lists. Examining the `task_struct` in `<linux/sched.h>`, we see two `struct list_head` objects:

```
children
```

and

```
sibling
```

These objects are pointers to a list of the task's children, as well as its siblings. Linux also maintains references to the `init` task (`struct task_struct init_task`). Using this information as well as macro operations on lists, we can iterate over the children of `init` as follows:

```
struct task_struct *task;
struct list_head *list;

list_for_each(list, &init_task->children) {
  task = list_entry(list, struct task_struct, sibling);
  /* task points to the next child in the list */
}
```

The `list_for_each()` macro is passed two parameters, both of type `struct list_head`:

- A pointer to the head of the list to be traversed
- A pointer to the head node of the list to be traversed

At each iteration of `list_for_each()`, the first parameter is set to the `list` structure of the next child. We then use this value to obtain each structure in the list using the `list_entry()` macro.

Part II Assignment

Beginning from the `init` task, design a kernel module that iterates over all tasks in the system using a DFS tree. Just as in the first part of this project, output the name, state, and pid of each task. Perform this iteration in the kernel entry module so that its output appears in the kernel log buffer.

If you output all tasks in the system, you may see many more tasks than appear with the `ps -ael` command. This is because some threads appear as children but do not show up as ordinary processes. Therefore, to check the output of the DFS tree, use the command

```
ps -eLf
```

This command lists all tasks—including threads—in the system. To verify that you have indeed performed an appropriate DFS iteration, you will have to examine the relationships among the various tasks output by the ps command.