Elec. 377 - Operating Systems
Lab 2 - Process Scheduling Spying

Lab Dates: Oct. 1/15, 2012, Due: Oct. 18, 2012

Objectives

• Based on lab 0's */proc* file system implementation module, retrieve information about the current state of the processes on the system.

Introduction

In this lab we're going to expose some of the kernel internals. As we've seen, when code is compiled into a module and loaded directly into the kernel, it has complete access to everything in the machine. It is the kernel that enforces security in user programs. In this lab, we're going to spy on the some of the scheduling information in the kernel and report the results to the user level through a */proc* file.
Each process has a *task_struct* that contains information about it. Information like the priority and PID of a process is stored here. When in a module, there exists a pointer to the calling process's *task_struct*, called *current*. In addition, the first process started on a Linux system is the *init* process. The *task_struct* for the init process is stored in a global kernel variable called *init_task*. The definition of the *task_struct* is in the file */usr/include/linux/sched.h*. We will also be looking at some global scheduling data. These are the number of running processes and the number of threads.

You are to write a linux module that provides a proc file that has two parts. An example output is shown at the end of this assignment. The first part is a header that contains two lines: the number of running tasks, and the number of threads. The second part of the file contains one line for each process, with each line containing the PID, the uid and the priority of the process.

This lab is going to require some research in the lab on your part. You are going to have to look at the sched.h header file, poke around in some /proc and system configuration files, and read some man pages. Much of the first lab should be devoted to research and planning. This includes both the planning for the design and the testing.

Statistics

There are two kernel global variables, both of type *int*, that contain the statistics that we are interested in. They are nr_running, nr_threads. If you look in sched.h you will find the external declarations for them. There is only one problem. The main purpose of /usr/include/linux is for building the kernel. There is a file in the kernel build which controls which of the variables are visible to modules at runtime. One of these variables is visible, the other is not. How do we find out? The file /proc/ksyms contains the list of visible kernel symbols and their addresses. Use the command *grep* to look for symbols in the file (do a 'man grep' to find out how to use grep).

If the symbol is not defined in the kernel, then how do we access the variable? This is where things get tricky!! The proper way to do this is to modify the file that controls the visible symbols and recompile the kernel making both symbols visible. Unfortunately, our vmware images are not large enough to do this. But if we knew where the variable was located in memory, we could read the memory directly. The file System.map in the directory boot contains the addresses of all symbols visible or not in your kernel. These are hexidecimal addresses. So code of the form:

```
        #define XXX_LOC 0xabcdef01
        …
        int *xxx = (int*)XXX_LOC;
```

could be used to access the integer stored at location 0xabcdef01. So your first part of research is to find out which of the two variables can be used directly in the module, and which must be accessed by address, and what the address of that variable is. Remember that hexidecimal constants in C are prefixed by '0x';

All Processes (Tasks)

It's easy to get the properties of the currently running task using *current* or the init task using *init_task*, but what about the others? The tasks are linked together in a circular linked list. The field *next_task* provides the address of the next task in the list. The priority of the task is given in the *nice* field.

Protocol for /proc files

The *eof* flag does not mean end of file. It actually means end of request. The kernel makes a call to the proc file system handler, which calls your function multiple times to fill up a buffer. You must return 0 bytes to indicate the end of the file. The *eof* flag signals the end a block of data to return to the calling program. Since the file position passed may be longer than the page, you also must indicate the buffer position using the start parameter. The signature for a read function is:

*int the_read_function(char *page, char **start, off_t fpos, int blen, int *eof, void * data).*

So in our case, what we will do, is write the data into the memory pointed to by page (e.g. sprintf(page,.....);), set *start to the beginning of the page (i.e. *start = page) and set *eof to 1 on each call to the read procedure. When we are at the end of the list of processes, we return 0 to indicate the end of the file.

As a reminder, remember that the code you are working on can crash your system. Save your files to the svn server before you insert your module into the kernel

Testing

Us the ps command to find the process priority (hint: look at the -o flag). You can also find out the number of running processes and threads from various system commands.

You do not have to hand in a prelab for this lab.

Suggested Program Structure

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linuxsched.h>

static stuct task_struct *firstTask, *nextTask;

int my_read_proc(...){
     if (file_pos == 0){
          // write the header to the buffer
          // find the first valid task (init_task)
     } else {
          // write task info to the buffer
          // advance the task pointer
     }
     if (file_pos > 0 && we are at beginning of list again){
          return 0;
     }
     *eof = 1;
     *start=page
     return numChars;
}
int init_module(){
     …
}
void cleanup_modul(){
     …
}
```

Sample Output (from user program using /proc file):

```
Number of running processes: 4
Nuumber of threads: 22
PID     UID     NICE
1       0       30
2       0       30
3       0       21
4       0       30
5       0       30
…
```