Assignment0: Linux Basics and /proc

CS314 Operating Systems

This project was adapted from Gary Nutt's Excellent Book "Kernel Projects for Linux" published by Addison-Wesley 2001.

You will learn several important characteristics of the Linux kernel, processes, memory, and other resources. You will write a program to use the /proc mechanism to inspect various kernel values that reflect the machine's load average, process resource utilization, and so on. After you have obtained the kernel status, you will prepare a report of the cumulative behavior that you observed.

Introduction

The Linux kernel is a collection of data structure instances (*kernel variables)* and functions. The collective kernel variables define the kernel's perspective of the state of the entire computer system. Each externally invoked function-a system call or an IRQ-provides a prescribed service and causes the system state to be changed by having the kernel code change its kernel variables. If you could inspect the kernel variables, then you could infer the state of the entire computer system.

Many variables make up the entire kernel state, including variables to represent each process, each open file, the memory, and the CPU. Kernel variables are no different than ordinary program variables, other than they are kept in kernel space. They can be allocated on a stack (in the kernel space) or in static memory so that they do not disappear when a stack frame is destroyed. Because the kernel is implemented as a monolithic module, many of the kernel variables are declared as global static variables. Some of the kernel variables are instances of built-in C/C++ data types, but most are instances of an extensible data type, a C/C++ struct.

In general, the extensible data structures are defined in C/C++ header files (also called `include` files or `.h` files) in the Linux source code tree.

A useful first step to understanding the details of Linux implementation is to explore the kernel variables and data structure definitions that are used to write the code. Much of the strategy for solving the problems in the remaining exercises is to focus on a particular aspect of the kernel, to study its data structures and functions, and then to solve the problem. The exercise introduction will help you to determine which parts of the source code tree to focus on for that exercise.

You have not learned enough to be able use a kernel debugger (or to write your own kernel extensions) to read the values stored in kernel variables. However, you can begin to examine some of those values by using existing tools. This exercise takes this latter approach to let you inspect the kernel state and to give you some intuition about how the kernel behaves.

Problem Statement

This exercise is to study some aspects of the organization and behavior of a Linux system by observing values stored in kernel variables.

Part A

Answer the following questions about the Linux machine that you will be using to do these exercises. If your school's lab has several machines, then choose one on which to base your answers.

- What is the CPU type and model?
- What version of the Linux kernel is being used?
- How long (in days, hours, and minutes) has it been since the system was last booted?
- How much of the total CPU time has been spent executing in user mode? System mode? Idle?
- How much memory is configured into it?
- How much memory is currently available on it?
- How many disk read/write requests have been made?
- How many context switches has the kernel performed?
- How many processes have been created since the system was booted?

Part B

Write a default version of program to report the behavior of the Linux kernel by inspecting kernel state. The program should print the following values to the user screen or console:

- CPU type and model
- Kernel version
- Amount of time since the system was last booted, in the form `dd:hh:mm:ss` (for example, 3 days 13 hours 46 minutes 32 seconds would be formatted as `03:13:46:32`)

Call your program "observer" and run it by typing the following at the linux shell prompt:

```
$ observer
```

Part C

Extend the program in Part A so that it also prints the following:

- The amount of time that the CPU has spent in user mode, in system mode, and idle
- The number of disk requests made on the system
- The number of context switches that the kernel has performed
- The time when the system was last booted
- The number of processes that have been created since the system was booted

Part D

Extend the program again so that it also prints the following:

- The amount of memory configured into this computer

- The amount of memory currently available
- A list of load averages (each averaged over the last minute)

This information will allow another program to plot these values against time so that a user can see how the load average varied over some time interval.

Allow the user to specify how the load average is sampled. To do this you need to provide two additional parameters:

- One to indicate how often the load average should be read from the kernel
- One to indicate the time interval over which the load average should be read

The user can provide this information to your program by typing the following at the linux shell prompt:

```
$observer –s 2 60
```

In this example the load average observation would run for 60 seconds, sampling the kernel table about once every 2 seconds. To observe the load on the system, you need to ensure that the computer is doing some work rather than simply running your program. For example, open and close windows, move windows around, and even run some programs in other windows.

Attacking the Problem

Linux, Solaris, and other versions of provide a UNIX very useful mechanism for inspecting the kernel state, called the **`/proc`** file system. This is the key mechanism that you can use to do this exercise.

One thing I found particularly helpful with this project was using the "man command" to get information about the "/proc" filesytem. "man" is short for "manual" and will give you all kinds of useful information about aspects of the operating system. Try typing "man proc".

The `/proc` File System

In the **`/proc`** system, the address space of another process can be accessed with file input and file output system calls, which allows a debugger to access a process being debugged with much greater efficiency. The page (or pages) of interest in the child process is mapped into the kernel address space. The requested data can then be copied directly from the kernel to the parent address space. In addition, /proc can be used to collect information about processes in the system.

The **`/proc`** file system is an OS mechanism whose interface appears as a directory in the conventional UNIX file system (in the root directory). You can change to /proc just as you change to any other directory.

For example:

```
bash$ cd /proc
```

makes **`/proc`** the current directory. Once you have made **`/proc`** the current directory, you can list its contents by using the **`ls`** command. The contents appear to be ordinary files and directories. However, a file in **`/proc`** or one of its subdirectories is actually a program that reads kernel variables and reports them as ASCII strings. Some of these routines read the kernel tables only when the pseudo file is opened, whereas others read the tables each time that the file read. Thus file input might behave differently than you expect, since they are not really operating on files at all.

The **`/proc`** implementation provided with Linux can read many different kernel tables. Several directories as well as files are contained in **`/proc`**. Each file reads one or more kernel variables, and the subdirectories with numeric names contain more pseudo files to read information about the process whose process ID is the same as the directory name. The directory **`self`** contains process-specific information for the process that is using /proc. The exact contents of the **`/proc`** directory tree vary among different Linux versions, so look at the documentation in the proc man page:

```
bash$ man proc
```

Files in **/proc** are read just like ordinary ASCII files. For example, when you type to the shell a command such as:

```
bash$ cat /proc/version
```

you will get a message printed to stdout (pronounced “standard out” this is the user’s screen... the same place you write to when you use the C++ cout object) that resembles the following:

```
Linux version 2.2.12 (gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)) #1 Mon
Sep 27 10:40:35 EDT 1999
```

To read a **`/proc`** pseudo file's contents, you open the file and then use the C++ extraction operator to read the file.

Using `argc` and `argv`

For Part D, the user might type this from the command line:

```
bash$ observer -s 10 600
```

where –s means to gather load averages and 10 600 means that the load averages are to be computed once every 10 seconds until 600 seconds have elapsed.

The following program demonstrates how to read command line arguments into your C++ program:

```
//
// This program demonstrates how to read command line
// arguments using argc and argv.
//
// argc - counts the number of space delimited arguments
// argv - pointer to an array of the space delimited arguments
//        the arguments are stored as C-Strings.  C-Strings
//        are an array of characters terminated with the null
//        character.
//
// For example, if this program is called "prog1", and you typed
// the following on the command line:
//
// > prog1 hello world 2
//
// Here's what the output of the program would be:
//
// argc is 4
// argv[0] is prog1 (array of characters 'p','r','o','g',’1’ and '\0')
// argv[1] is hello
// argv[2] is world
// argv[3] is 2
//
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
      cout << "argc is " << argc << endl;

      for (int i=0; i<argc; i++)
      {
            cout << "argv[" << i << "] is "
                   << argv[i] << endl;
      }

      return 0;

}
```

Organizing a Solution

For Parts B, C and D your programs must have different arguments on the command line. Therefore one of your first actions should be to parse the command line with which the program is called so as to determine the shell parameters being passed to it via the argv array.

You finish initializing by getting the current time of day and printing a greeting that includes the name of the machine that you are inspecting.

```
// Get Time of Day
struct timeval now;
gettimeofday(&now, NULL);
cout << "Status report as of : "
     << ctime((time_t *) &now.tv_sec) << endl;

// Print machine name
in.open("/proc/sys/kernel/hostname");
string s;
in >> s;
cout << "Machine name: " << s << endl;

in.close();
```

Now you are ready to do the work, that is, to start reading kernel variables by using various **/proc** files. The previous code segment contains an example of how to read the **/proc/sys/kernel/hostname** file. You can use it as a prototype for solving the exercise by reading other pseudo files. This will require some exploration of **/proc** and inspection of the various pseudo files as you investigate different directories.

In Part D, you are to compute a load average. For this, your code needs to sleep for a while, wake up, sample the current load average, and then go back to sleep. Here is a code fragment that will accomplish that work.

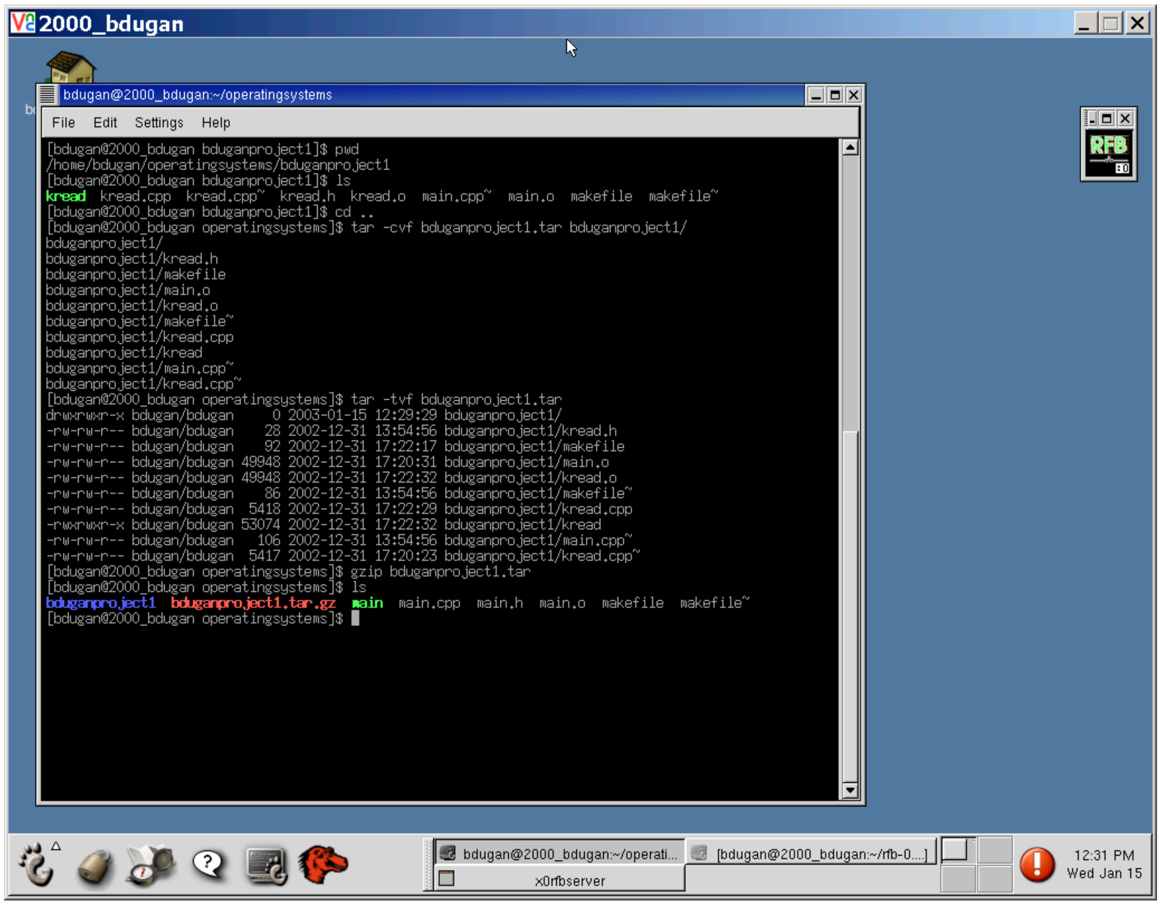
```
// Compute load average with periodic samples
int iteration=0;
while (iteration < duration)
{
 sleep(interval);
 sampleLoadAverage();
 iteration = iteration + interval;
}
```

DELIVERABLES (Please follow these directions precisely… it saves me a lot of time when I grade your assignment):

Upload to elearn the following:

- An ELECTRONIC document containing the answers to Part A. Call this document SOLUTIONS.TXT. In addition to the answer, you must show me how you got there.
- All the files necessary to compile, link, and run your program solutions to Part B, C, and D.
- An ELECTRONIC document describing how to run the three programs you created. Call this document README.TXT.
- These files should be placed in a directory called "`<username>assignment0`".

- Use the tar command to place all the files in a single file called `"<username>assignment0.tar"`. Assuming you are in the directory `"<username>assignment0"` do the following:
    - Goto the parent directory: `cd ..`
    - tar the files : `tar -cvf <username>project0.tar ./<username>assignment0`
    - Verify the files have been placed in a tar file : `tar -tvf <username>assignment0.tar`
    - Compress the files using gzip: `gzip <username> assignment0.tar`
    - Verify that the gzipped file exists: `ls <username> assignment0.tar.gz`
- Here's a screen snapshot (just replace assignment0 with project1) of these commands:


2000_bdugan
bdugan@2000_bdugan:~/operatingsystems
File
Edit
Settings
Help
bdugan@2000_bdugan:~/operati...
[bdugan@2000_bdugan:~/rfb-0....]
x0rfbserver
12:31 PM
Wed Jan 15

```
[bdugan@2000_bdugan bduganproject1]$ pwd
/home/bdugan/operatingsystems/bduganproject1
[bdugan@2000_bdugan bduganproject1]$ ls
kread  kread.cpp  kread.cpp~  kread.h  kread.o  main.cpp~  main.o  makefile  makefile~
[bdugan@2000_bdugan bduganproject1]$ cd ..
[bdugan@2000_bdugan operatingsystems]$ tar -cvf bduganproject1.tar bduganproject1/
bduganproject1/
bduganproject1/kread.h
bduganproject1/makefile
bduganproject1/main.o
bduganproject1/kread.o
bduganproject1/makefile~
bduganproject1/kread.cpp
bduganproject1/kread
bduganproject1/main.cpp~
bduganproject1/kread.cpp~
[bdugan@2000_bdugan operatingsystems]$ tar -tvf bduganproject1.tar
drwxrwxr-x bdugan/bdugan     0 2003-01-15 12:29:29 bduganproject1/
-rw-rw-r-- bdugan/bdugan    28 2002-12-31 13:54:56 bduganproject1/kread.h
-rw-rw-r-- bdugan/bdugan    92 2002-12-31 17:22:17 bduganproject1/makefile
-rw-rw-r-- bdugan/bdugan 49948 2002-12-31 17:20:31 bduganproject1/main.o
-rw-rw-r-- bdugan/bdugan 49948 2002-12-31 17:22:32 bduganproject1/kread.o
-rw-rw-r-- bdugan/bdugan    86 2002-12-31 13:54:56 bduganproject1/makefile~
-rw-rw-r-- bdugan/bdugan  5418 2002-12-31 17:22:29 bduganproject1/kread.cpp
-rwxrwxr-x bdugan/bdugan 53074 2002-12-31 17:22:32 bduganproject1/kread
-rw-rw-r-- bdugan/bdugan   106 2002-12-31 13:54:56 bduganproject1/main.cpp~
-rw-rw-r-- bdugan/bdugan  5417 2002-12-31 17:20:23 bduganproject1/kread.cpp~
[bdugan@2000_bdugan operatingsystems]$ gzip bduganproject1.tar
[bdugan@2000_bdugan operatingsystems]$ ls
bduganproject1  bduganproject1.tar.gz  main  main.cpp  main.h  main.o  makefile  makefile~
[bdugan@2000_bdugan operatingsystems]$ 
```