Assignment 1: Observing Linux Behavior

CSC 139 Operating System Principles - Fall 2018

Posted on Sep. 7, due on Sep. 20 (11:59 pm)

This assignment was adapted from Gary Nutt's book *Kernel Projects for Linux* published by Addison-Wesley, 2001.

1 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 interrupt request - provides a prescribed service and causes the system state to be changed by having the kernel code changed 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 C program variables, other than they are kept in kernel space. They can be allocated on a stack (in 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 data types, but most are instances of an extensible data type, a C struct.

The /proc filesystem is a virtual filesystem that permits a novel approach for communication between the Linux kernel and user space. The /proc files system mounted under the /proc directory. It is a virtual file system that provides an interface to kernel data structures in a form that looks like files and directory's on a file system. In the /proc filesystem, virtual files can be read from or written to as a means of communicating with entities in the kernel, but unlike regular files, the content of these virtual files is dynamically created. It's sometimes referred to as a process information pseudo-file system. It doesn't contain 'real' files but runtime system information (e.g. system memory, devices mounted, hardware configuration, etc.). The /proc file system provides an easy mechanism for viewing and changing various system attributes. For this reason it can be regarded as a control and information center for the kernel. In fact, quite a lot of system utilities are simply calls to files in this directory. For example, 'lsmod' is the same as 'cat /proc/modules' while 'lspci' is a synonym for 'cat /proc/pci'. By altering files located in this directory you can even read/change kernel parameters (sysctl) while the system is running.

2 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. 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.

2.1 Part A

Answer the following questions about the Linux machine that you will be using to do these exercises. If you will use ECS servers, 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 disks 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?

2.2 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`)
- report date (gettimeofday) and machine hostname

2.3 Part C

Write a second version of the program in Part B that prints the same information as the default version plus the following:

- The amount of time 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

2.4 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 will need 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 first version of your program might be called by `observer` and the second version by `observer -s`. Then the third version could be called by `observer -l 2 60`, whereby 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.

3 Attacking the Problem

Linux, Solaris, and other versions of UNIX provide a 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.

3.1 The /proc File 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 is read. Thus the various read functions 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 read 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. To read /proc pseudo file's contents, you open the file and then use the stdio library routines such as `fgets()` or `fscanf()` to read the file. 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

```
cat /proc/version
```

you will get a message printed to stdout 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
```

3.2 Using argc and argv

If you do Part C or D in addition to Part B, you will need to pass parameters to your program from the shell. For example, suppose that your solution program is named `observer`. To solve Part B, you could call it with

```
$ observer
```

whereas to solve Part C or D, you could call with

```
$ observer -s
```

If the program is providing information required for Part D, it might be called by

```
$ observer -l 10 600
```

computing information once every 10 seconds until 600 seconds have elapsed.

The following code segment is an example for handling the three different ways that `observer` can be called. A C program may have the header file of the form:

```
int main(int argc, char *argv[])
```

You may omit the two arguments, **argc** and **argv** if no parameters are to be passed to the program by the shell. Alternately, they can be initialized so that **argc** is the number of symbols on the command line and **argv** is an array of pointers to character strings symbols on the command line. For example, if the `observer` program is called with no parameters, then **argc** will be set to 1 and **argv[0]** will point to the string `observer`. In the second example, **argc** will be set to 2, with **argv[0]** pointing to the string `observer` and **argv[1]** pointing to the string `-s`.

The C main program can now reference these arguments as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
...
int main(int argc, char *argv[]) {
   ...
   char repTypeName[16];
   ...
   /* Determine report type */
   reportType = STANDARD;
```

```
    strcpy(repTypeName, "Standard");
    if (argc > 1) {
       sscanf(argv[1], "%c%c", &c1, &c2);
    if (c1 != '-') {
       fprintf(stderr, "usage: observer [-s][-l int dur]\n");
          exit(1);
    }
    if (c2 == 's') {
       reportType = SHORT;
       strcpy(repTypeName, "Short");
    }
    if (c2 == 'l') {
       reportType = LONG;
       strcpy(repTypeName, "Long");
          interval = atoi(argv[2]);
          duration = atoi(argv[3]);
    }
 }
...
}
```

3.3 Organizing a Solution

For Part 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 you are inspecting:

```
#include <unistd.h>
...
FILE *thisProcFile;

/* Finish initializing ... */
gettimeofday(&now, NULL); // Get the current time
printf("Status report type %s at %s\n", repTypeName, ctime(&(now.tv_sec)));

// Get the host filename and print it
thisProcFile = fopen (" /proc/sys/kernel/hostname", "r");
fgets(lineBuf, LB_SIZE+1, thisProcFile);
printf("Machine hostname: %s", lineBuf);

fclose(thisProcFile);
```

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/svs/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 the /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.

```
#include <sys/time.h>
#include <time.h>
/* some initialization ... */
struct timeval now;
/* Finish initializing ... */
while (iteration < duration) {
   sleep(interval);
   sampleLoadAvg();
   iteration += interval);
}
```

Now you are ready to create the entire solution:

```
#include <stdio.h>
#include <sys/time.h>
...
int main(int argc, char *argv[]) {
...
   char repTypeName[16];
...
   /* Determine report type */
   reportType = STANDARD;
   strcpy(repTypeName, "Standard");
   if (argc > 1) {
      sscanf(argv[1], "%c%c", &c1, &c2);
      if (c1 != '-') {
         fprintf(stderr, "usage: observer [-s][-l int dur]\n");
         exit(1);
      }
      if (c2 == 's') {
         reportType = SHORT;
         strcpy(repTypeName, "Short");
      }
      if (c2 == 'l') {
         reportType = LONG;
         strcpy(repTypeName, "Long");
         interval = atoi(argv[2]);
         duration = atoi(argv[3]);
      }
   }

   /* Finish initialization */
   gettimeofday(&now, NULL); // Get the current time
   printf("Status report type %s at %s\n", repTypeName, ctime(&(now.tv_sec)));

   /* Get the host filename and print it */
   thisProcFile = fopen("/proc/sys/kernel/hostname", "r");
   fgets(lineBuf, LB_SIZE+1, thisProcFile);
   printf("Machine hostname: %s", lineBuf);
   fclose(thsProcFile);

   /* Code to read the relevant /proc files */
   ...
```

```
   while (iteration < duration) {
      sleep(interval);
      sampleLoadAvg();
      iteration += interval);
   }
   exit 0;
}
```

4 Deliverables

Upload to Canvas the following:

- A 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.
- A document describing how to run the three programs you created. Call this document `README.TXT`.
- These files should be placed in a directory called `<ECS username>-asgmt1`.
- Use tar command to place all the files in a single file called `<ECS username>-asgmt1.tar`. Assuming you are in the directory `<ECS username>-asgmt1` do the following:
  - Goto the parent directory: `cd ..`
  - tar the files:
    `tar -cvf <ECS username>-asgmt1.tar ./<ECS username>-asgmt1`
  - Verify the files have been placed in a tar file:
    `tar -tvf <ECS username>-asgmt1.tar`
  - Compress the files using gzip: `gzip <ECS username>-asgmt1.tar`
  - Verify that the gzipped file exists: `ls <ECS username>-asgmt1.tar.gz`