Using the /proc Filesystem

Michael Jantz
Prasad Kulkarni
Greeshma Umapathi
Douglas Niehaus

Introduction

- This lab introduces the /proc virtual file system in Linux.
- This lab assumes you understand the use of PThreads and the dining philosopher's problem. Thus, you will need to have completed the lab on the dining philosopher's problem before proceeding with this lab.
- Go ahead and make and tag the starter code for this lab:

```
bash>cd eecs678-procfs-lab/; make; ctags -R
```

The *ps* Command

- Linux distributions usually come with several programs for monitoring system activity.
- The *ps* command displays a panoply of information about the processes on the system. The *ux* argument will print information for every process the user who invoked it has started. Adding *a* (to give '*ps aux*' will print information for all processes)*:*

-bash-3.2$ ps ux

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
mjantz	19323	0.0	0.0	13528	2108	?	S	18:11	0:02	sshd: mjantz@pts/0
mjantz	19324	0.0	0.0	5100	1532	pts/0	Ss	18:11	0:00	-bash
mjantz	21248	0.0	0.0	4760	864	pts/0	R+	22:08	0:00	ps ux

- Shown here are the processes for the user *mjantz*:
  - %CPU – the CPU utilization of the process (this is currently the ratio of time spent on the CPU over real time since process was started)
  - %MEM – ratio of the process' resident set size to the physical memory on the machine.
  - VSZ – Virtual memory size of the process in memory (in KB).
  - RSS – The resident set size for this process in KB.
  - START / TIME – The start and total running times of the process.
  - COMMAND – The command invoked to start the process.

top

- *top* is an interactive version of *ps*. Run *top -u username* with your username specified as *username*:

```
top - 23:04:58 up 155 days,  2:30,  1 user,  load average: 0.00, 0.02, 0.00
Tasks: 157 total,   1 running, 156 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   3374632k total,  1876212k used,  1498420k free,   179676k buffers
Swap:  2096440k total,    59276k used,  2037164k free,  1463380k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
21539 mjantz    20   0  2528  984  748 R  0.3  0.0   0:00.13 top
21502 mjantz    20   0 13376 1908 1084 S  0.0  0.1   0:00.00 sshd
21503 mjantz    20   0  5096 1472 1196 S  0.0  0.0   0:00.02 bash
```

- Again, you will see a variety of statistics.
- This program is interactive and does not immediately return you to the shell. Every so often (3 seconds by default), the printout refreshes its ratios and statistics to reflect values since the last screen refresh.
- You can select different columns to sort on and display additional statistics. Type 'h' to get a listing of commands. 'q' will quit the program.
- See the man page (*man top*) for an explanation of all these statistics.

/proc

- *ps* and *top* are user level processes (we forked them from a shell in user land).
- However, they display information about processes that requires a view of the entire system. In the case of *top*, this information may be updated frequently. Thus, many OS-level structures need to expose information to user programs.
- How do systems provide this type of information to user-level processes?
  - Event logging and / or tracing – requires extra space on disk and overhead to create log files
  - Direct access to kernel memory – requires user-level programmers to have knowledge of kernel data structures (which are not standardized and may change)
- Linux systems choose to provide this information through a hierarchical file-like structure called the */proc virtual filesystem*. /proc allows user-level programs to access information about processes and other system information in a convenient and standardized way.

A Virtual File System

- The /proc file system uses the file model of representing data to present dynamically changing aspects of process state
- Many files in /proc are created at boot time and destroyed at shut down
  - Others come and go as processes live and die
- They never actually exist on a physical medium except, arguably, the RAM.
  - In this sense, /proc is a virtual file system.
- Because the data is represented using the file model, we can use the same interface we use with regular files to interact with /proc.

Listing Files

- The /proc directory contains several important top level files.
- We can view these using the same shell commands we use for viewing regular files. For instance, try 'ls /proc/'.
- You should see a directory filled with several regular files and several directories (your shell should color code directories differently than regular files). You can glean more information from the *ls* command using the -l option:

```
-bash-3.2$ ls -l /proc/cpuinfo
-r--r--r-- 1 root root 0 2009-04-28 09:04 /proc/cpuinfo
```

- This information means /proc/cpuinfo:
  - Is a regular file (if it were a directory – we would see a 'd' in the first slot of -r--r--r-- instead of a -).
  - Has *permissions* for the owner, group, and user to read the file (write and execute are not available for anybody).
  - Contains 1 hard link (itself). This would be more interesting if /proc/cpuinfo were a directory.
  - Is owned by root and is part of the group root.
  - Has a size of 0 on the disk. This is because /proc/cpuinfo is a virtual file.
  - Has a timestamp of April 28, 2009 at 09:04. This should be recent as this file is constantly updated.
  - Is located at /proc/cpuinfo

Viewing Top-Level Files

- Since /proc/cpuinfo is a read-only file, it doesn't make much sense to do anything other than read it. Go ahead and use *cat* to print out the contents of /proc/cpuinfo:

```
-bash-3.2$ cat /proc/cpuinfo
```

- A wave of text should cascade across your screen. This text gives you, for every processor in the system, a variety of statistics. For instance, this text:

```
processor: 7
vendor_id: GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Xeon(R) CPU          X5355  @ 2.66GHz
stepping  : 7
cpu MHz         : 2659.987
cache size      : 4096 KB
```

- Tells you that (among other things) the 8th processor on this machine (processor ID's are indexed at 0), is an Intel Xeon X5355 processor clocked at 2.66GHz. It's observed running speed (@ boot time) is 2.659GHz and it is equipped with a 4MB cache.

Other Important Files

- There are several other files in /proc that might be interesting to explore on your own
- We recommend /proc/meminfo, /proc/net/, and /proc/uptime as files a curious observer might find interesting
- Exploring these files pretty much requires use of the /proc documentation in the /proc manual (man proc)
- However, similar to *ps* and *top*, they all have built in commands (*free, netstat*, and *uptime*) for pretty printing their contents
- For the purposes of this lab, we will ignore these other top-level files and focus instead on the several numbered directories which represent state of processes

Process Directories

- Each numbered directory corresponds to a process running on the system
  - The number is the process PID
- Within each directory, there are several more files and directories containing data the kernel has allocated for the running process.
- Find the pid of your shell process (use 'ps ux'), and list the contents of that directory in /proc:

```
-bash-3.2$ ls /proc/23831/
attr  cgroup      cmdline          cpuset  environ  fd      io       limits    maps  mountinfo  mountstats  oom_adj    pagemap  sched      sessionid  stat   status   task
auxv  clear_refs  coredump_filter  cwd     exe      fdinfo  latency  loginuid  mem   mounts     net         oom_score  root     schedstat  smaps      statm  syscall  wchan
-bash-3.2$
```

- Again, we find several files and directories containing a variety of statistics.
- Many of these are self explanatory, and those that aren't can be looked up using the /proc manual page. This lab is too short (and it would be a waste of everybody's time) to cover everything in these directories.
- We will cover a couple files, however, in these process directories that give you a glimpse of how powerful using /proc can be.

Using /proc in Your Programs

- For the rest of this lab, we will revisit the dining philosopher's problem to show how /proc can be used as a programming tool.
- In an earlier lab we created a program that created threads that alternated between states of thinking and eating. The main thread monitored the progress of these threads and killed all threads if deadlock was ever reached.
- Each thread modified state variables every time they ate to show they had indeed progressed.
- This solution is OK (and is probably the most portable), but if we didn't care about portability, we could use the statistics gathered by /proc to determine, in a more direct way, whether or not our threads have made progress.
- The starter code for this lab implements the dining philosopher's problem, but the main thread's check_for_deadlock() function has been left as an empty loop. We will implement this using /proc in today's lab.

The Stat File

- Each /proc/<PID>/ process state directory contains a file called 'stat'.
- This file contains several miscellaneous statistics about the running program that is easy for other programs to parse. The file 'status' contains much of the same information in a human readable form.
  - *procstat* (usage: ./procstat PID) prints a human-readable version of this file for a given PID
- For the purposes of this lab, we will focus on two entries in this file, the running time (in jiffies – the Linux timer tick) the process has spent executing in *user mode*, and the running time (in jiffies) the process has spent executing in *system mode*.
- If we could periodically gather these times for each of the threads the main thread is monitoring, we could test if they've changed since the last time we looked at them, and if they haven't, we probably have deadlock.

Opening My Threads' Stat Files

- Each process directory in /proc contains a special link called **self/** which allows a process to refer to its /proc entry without knowing its own PID.
- Also, each multithreaded process' /proc directory contains a directory called **task/**, which includes subdirectories for each of its threads.
- /proc/self/task numbers its directories by the thread ID of each thread in the process group, in a similar manner as the /proc directory, only using thread IDs as opposed to PIDs.
- We can get each thread ID using the **gettid()** call provided because getting this to work was a little subtle
  - Basically the **gettid()** system call is not portable and, thus, we have to wrap it in this syscall() function – I've done this for you at the top of each philosopher thread.
- Now, we can form a filename for each of the stat files of the threads we wish to monitor by using the **sprintf( )** library call with the string: “/proc/self/task/%d/stat”, and providing the i'th diner's thread id by passing diners[i].tid as the final argument.

Scanning to the Data We Want

- Our approach is to scan this file until we reach the data we want.
- After a quick reference of man proc, we see that user and system times are actually the 14th and 15th fields of the stat file.
- You will need to use fopen() to open a file stream to each thread's stat file. Consider using the (currently) unused variables I've declared for you at the top of check_for_deadlock().
- Next, you can use fscanf calls to scan over the first 13 fields of data and finally read the 14th and 15th fields into appropriate locations.
- The compiler should warn you if you are missing or have ill formed arguments to your fscanf call(s).

fscanf( )

- int fscanf(FILE *stream, const char *format, ...);
- fscanf: works just like scanf, but 1st parameter is a file variable. The fscanf() function reads from the named input ***stream***.
- sscanf does the same thing while reading strings
- By using an optional character * in the format specifier, fscanf() reads input as directed by the conversion specification, but discards the input.
- An example for this is – fscanf(filep, “%*s”); will ignore one string field that is read from ***filep.***

Checking For Deadlock

- Once you have the times stored away in local variables, you can perform a check for deadlock.
- A global symbol "DEADLOCK" controls how the program compiles.
  - A value of zero compiles in code avoiding deadlock, nonzero deadlocks. The code as provided sets DEADLOCK to 1.
- ACTIVE_DURATION controls how long eating and thinking take
- The check_for_deadlock() routine is incomplete, but the structure assumes deadlock has occurred and looks for evidence of progress.
- This is where you should think about how the user and system time values can represent progress. As a hint, you need to use the user and system time arrays to check for deadlock. The progress arrays should only be updated for printing later.
- Finally, when you are through, be sure to close the open file stream with *fclose()*.

Final Output

- When you are through, you should ensure that your solution works in situations where deadlock occurs, and when it does not occur.
- By default, the starter code does deadlock (DEADLOCK = 1) bigger values of ACTIVE_DURATION make deadlock less likely, smaller make it more likely
- You can build a version which will avoid deadlock using the asymmetric solution by setting DEADLOCK = 0
- You can make and run the **dine** executable with the following command -
    - bash$ make test1
- When you are through, your output should resemble the following:

Philosopher:	P0	P1	P2	P3	P4
User time:	245 / 245	238 / 238	212 / 212	220 / 220	217 / 217
System time:	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
User time:	205 / 450	204 / 442	180 / 392	183 / 403	179 / 396
System time:	1 / 1	1 / 1	1 / 1	0 / 0	1 / 1

- If you're curious, the time units are in jiffies, the Linux timer tick unit, which is a parameter that is configurable at kernel compile time.