

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS  
Information and Computer Science Department  
ICS 431 OPERATING SYSTEMS  
Lab# 10

**Observing Operating System Behavior using /proc**

---

## Objective:

To study how we can observe the behavior of Linux kernel using /proc utility.

---

## Observing Operating System Behavior (using /proc):

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.

---

### What is a /proc File System? :

*The /proc file system isn't a file system in the standard sense. Rather, the /proc file system is an interface to the address space of running processes. With /proc, you can use standard UNIX system calls (e.g., open(), read(), write(), and ioctl()) to query or manipulate the processes' address space.* In fact, the Solaris ps(1) command uses /proc to determine the status of the processes listed in the process table.

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,

```
> 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 large files found in /proc are the address spaces of running processes--not standard UNIX files. Each filename is, in fact, just the PID of the running processes.* The owner and group owner of each file are the real-UID and primary group of the process's owner. The permission bits control access, as with any UNIX file. The most confusing part--the file's size--is actually quite simple: It's the total amount of memory (image size) the process uses. This amount doesn't represent actual bytes on disk, so you're not losing precious disk space to the files under /proc. So no, don't delete those files!.

### A sample listing from /proc.

```
redhat> ls -l /proc
total 69
dr-xr-xr-x   3 root    root0 Sep 10 01:35 1
dr-xr-xr-x   3 root    root   0 Sep 10 01:35 10314
```

```

dr-xr-xr-x    3 root    root      0 Sep 10 01:35 10315
dr-xr-xr-x    3 root    root      0 Sep 10 01:35 10317
dr-xr-xr-x    3 root    root      0 Sep 10 01:35 10318
dr-xr-xr-x    3 root    root      0 Sep 10 01:35 10320
dr-xr-xr-x    3 root    gradics   0 Sep 10 01:35 10337
dr-xr-xr-x    3 root    gradics   0 Sep 10 01:35 10340
lrwxrwxrwx    1 root    root      64 Sep  9 09:34 self -> 10364
-rw-r--r--    1 root    root      0 Sep 10 01:35 info
-r--r--r--    1 root    root      0 Sep 10 01:35 stat
-r--r--r--    1 root    root      0 Sep 10 01:35 swaps
dr-xr-xr-x   10 root    root      0 Sep 10 01:35 sys
dr-xr-xr-x    2 root    root      0 Sep 10 01:35 sysvipc
dr-xr-xr-x    4 root    root      0 Sep 10 01:35 tty
-r--r--r--    1 root    root      0 Sep 10 01:35 uptime
-r--r--r--    1 root    root      0 Sep 10 01:35 version

```

The method for manipulating files under `/proc` is the same as for ordinary UNIX files. All of your favorite system calls will work, including `ioctl()`. In the kernel, the vnode operations performed on a file under `/proc` are for procfs. This means that the actual vnode access system calls (e.g., `lookuppn()`) are eventually passed to procfs-savvy system calls (e.g., `prlookup()`).

---

## Contents of `/proc` File System:

The `proc` file system acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain kernel parameters at runtime.

First, we'll take a look at the read-only parts of `/proc`.

The output of the directory listing of the `/proc` directory is as follows.

```
redhat> ls /proc
```

```

1      11791 2612  448   657   806   901      fs      meminfo   sys
10314  11792 2613  453   670   834   906      ide     misc
      sysvipc
10315  11795 2614  467   694   860   apm      interrupts modules   tty
10317  2      2615  482    7     885   bus      iomem    mounts
      uptime
10318  2249  2616  5 7    12    886   cmdline  ioports  mtrr
      version
10320  2250  2624  571   727   887   cpuinfo  irq      net
10337  2253  2638  589   73    888   devices  kcore    partitions
10340  2280  2639  591   747   889   dma      kmsg     pci
10343  23092 2640  595   772   890   driver   ksyms    self
10362  23093 2804  596   787   891   execdomains loadavg  slabinfo
11786  2610  3     597   799   898   fb       locks    stat
11790  2611  4     6     8     899   filesystems mdstat   swaps

```

*The contents appear to be ordinary files and directories. However, a file in a `/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.

Every number and word that you see in the above snapshot are the contents of the `/proc` directory.

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.

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

```
redhat> cat /proc/version
```

you will get a message printed to screen that resembles the following

```
Linux version 2.4.2-2 (root@porky.devel.redhat.com) (gcc version
2.96 20000731 ( Red Hat Linux 7.1 2.96-79)) #1 Sun Apr 8
20:41:30 EDT 2001
```

To read a /proc pseudo file's contents, you open the file and then use stdio library routines such as fgets ( ) or fscanf ( ) to read a file.

The various directories that you see out here are the processes that were running on our machine at the instant we took a snapshot of the /proc file system.

---

## Process-Specific Subdirectories:

The directory /proc contains (among other things) one subdirectory for each process running on the system, which is named after the process ID (PID). The link [self](#) points to the process reading the file system. Each process subdirectory has the entries listed in Table 1-1.

**Table 1-1 Process specific entries in /proc:**

File	Content
<a href="#">cmdline</a>	Command line arguments
<a href="#">environ</a>	Values of environment variables
<a href="#">fd</a>	Directory, which contains all file descriptors
<a href="#">mem</a>	Memory held by this process
<a href="#">stat</a>	Process status
<a href="#">status</a>	Process status in human readable form
<a href="#">cwd</a>	Link to the current working directory
<a href="#">exe</a>	Link to the executable of this process
<a href="#">maps</a>	Memory maps
<a href="#">root</a>	Link to the root directory of this process
<a href="#">statm</a>	Process memory status information

## Examples:

"[status](#)", this file gives information regarding the [name of the process](#), its [current status](#), i.e [sleeping or awake](#), its [PID](#), its [UID](#), its [PPID](#) and a lot of other general information. This information can be viewed in a simpler and structured manner by using tools like, "ps" and "top".

For example, to get the status information of a process, all you have to do is read the file `/proc/PID/status`:

```
>cat /proc/self/status
Name: cat
State: R (running)
Pid: 5452
PPid: 743
Uid: 501 501 501 501
Gid: 100 100 100 100
Groups: 100 14 16
VmSize: 1112 kB
VmLck: 0 kB
VmRSS: 348 kB
VmData: 24 kB
VmStk: 12 kB
VmExe: 8 kB
VmLib: 1044 kB
SigPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 00000000fffffeff
CapPrm: 0000000000000000
CapEff: 0000000000000000
```

This shows you nearly the same information you would get if you viewed it with the `ps` command. In fact, `ps` uses the `proc` file system to obtain its information. The `statm` file contains more detailed information about the process memory usage. Its seven fields are explained in Table 1-2 .

**Table 1-2 Contents of the statm files:**

File	Content
size	total program size
resident	size of memory portions
shared	number of pages that are shared
trs	number of pages that are 'code'
drs	number of pages of data/stack
lrs	number of pages of library
dt	number of dirty pages

---

## Kernel Data:

Similar to the process entries, the kernel data files give information about the running kernel. The files used to obtain this information are contained in [/proc](#) and are listed in Table 1-3 . Not all of these will be present in your system. It depends on the kernel configuration and the loaded modules, which files are there, and which are missing.

**Table 1-3 Kernel info in /proc**

File	Content
apm	Advanced power management info
bus	Directory containing bus specific information
cmdline	Kernel command line
cpuinfo	Info about the CPU
devices	Available devices (block and character)
dma	Used DMS channels
filesystems	Supported filesystems
ide	Directory containing info about the IDE subsystem
interrupts	Interrupt usage
ioports	I/O port usage
kcore	Kernel core image
kmsg	Kernel messages
ksyms	Kernel symbol table
loadavg	Load average
locks	Kernel locks
meminfo	Memory info
misc	Miscellaneous
modules	List of loaded modules
mount	Mounted filesystems
net	Networking info
partitions	Table of partitions known to the system
rtc	Real time clock
scsi	SCSI info
slabinfo	Slab pool info
stat	Overall statistics
swaps	Swap space utilization
uptime	System uptime
version	Kernel version

You can, for example, check which interrupts are currently in use and what they are used for by looking in the file [/proc/interrupts](#):

```
> cat /proc/interrupts
CPU0
0: 8728810 XT-PIC timer
1: 895 XT-PIC keyboard
2: 0 XT-PIC cascade
3: 531695 XT-PIC aha152x
4: 2014133 XT-PIC serial
5: 44401 XT-PIC pcnet_cs
8: 2 XT-PIC rtc
11: 8 XT-PIC i82365
12: 182918 XT-PIC PS/2 Mouse
13: 1 XT-PIC fpu
14: 1232265 XT-PIC ide0
15: 7 XT-PIC ide1
NMI: 0
```

There are three more important subdirectories in [/proc](#): [net](#), [scsi](#), and [sys](#). The general rule is that the contents, or even the existence of these directories, depend on your kernel configuration. If SCSI is not enabled, the directory [scsi](#) may not exist. The same is true with the [net](#), which is there only when networking support is present in the running kernel.

The [slabinfo](#) file gives information about memory usage at the slab level. Linux uses slab pools for memory management above page level in version 2.2. Commonly used objects have their own slab pool (such as network buffers, directory cache).

---

## Generic information about the various [/proc/\\*](#) directories

### [/proc/cpuinfo](#)

Information about the processor, such as its type, make, model, and performance.

### [/proc/devices](#)

List of device drivers configured into the currently running kernel.

### [/proc/dma](#)

Shows which DMA channels are being used at the moment.

### [/proc/filesystems](#)

Filesystems configured into the kernel.

### [/proc/interrupts](#)

Shows which interrupts are in use, and how many of each there have been.

### [/proc/ioports](#)

Which I/O ports are in use at the moment.

---

## What [/proc](#) can tell me?

The tools bundled with Solaris that use [/proc](#) are quite informative. Found under [/usr/proc/bin](#), the tools provide a handy way of accessing critical bits of data about a given process.

The [proc](#) tools are utilities that exercise features of [/proc](#). Most of them take a list of process-ids (pid); those that do also accept [/proc/nnn](#) as a process-id, so the shell expansion [/proc/\\*](#) can be used to specify all processes in the system.

**pflags** Print the /proc **tracing flags**, the **pending** and **held signals**, and other /proc status information for each lwp in each process.

**pcred** Print the **credentials** (effective, real, saved UIDs and GIDs) of each process.

**pmap** Print the **address space map** of each process.

**pfiles** Report fstat(2) and fcntl(2) information for all open files in each process.

**pldd** List the **dynamic libraries** linked into each process, including shared objects explicitly attached using dlopen(3X). See also ldd(1).

**psig** List the **signal actions** of each process.

**pstack** Print a hex+symbolic **stack trace** for each lwp in each process.

**pwdx** Print the current working directory of each process.

**pstop** **Stop** each process (PR\_REQUESTED stop).

**prun** Set each process **running** (inverse of pstop ).

**pwait** **Wait** for all of the specified processes to terminate.

**ptree** Print the **process trees** containing the specified **pids or users**, with child processes indented from their respective parent processes. An argument of all digits is taken to be a process-id, otherwise it is assumed to be a user login name. Default is all processes.

**ptime** **Time** the command, like time(1), but using micro-state accounting for reproducible precision. Unlike time(1), children of the command are not timed.

```
vlsi> sleep 500 &  
[1] 15524
```

```
vlsi> ps  
PID TT S TIME COMMAND  
15394 pts/4 S 0:00 -csh  
15524 pts/4 S 0:00 sleep 500
```

For example, let's say you want to know how many files a process has open. You could use /usr/proc/bin/pfiles to help you. Using the pfiles command.

```
vlsi> pfiles 15524  
8619: -csh  
Current rlimit: 64 file descriptors  
0: S_IFCHR mode:0666 dev:32,0 ino:11466 uid:0 gid:3 rdev:13,2  
O_RDONLY|O_LARGEFILE  
1: S_IFCHR mode:0666 dev:32,0 ino:11466 uid:0 gid:3 rdev:13,2  
etc.
```

```
vlsi> pcred 15524  
15524: e/r/suid=2041 e/r/sgid=500  
vlsi> pwdx 15524  
15524: /usr/proc/bin  
vlsi> ptime 15524  
ptime: exec failed  
real 0.015  
user 0.000  
sys 0.008
```

```
vlsi> ptree 15524
169 /usr/sbin/inetd -s
15392 in.rlogind
15394 -csh
15524 sleep 500
```

As shown in the code, using any of the /proc commands is as simple as the command name followed by the PID. *Keep those permission bits in mind, however!* As with all UNIX files, you'll be unable to access the process data for a given PID if you don't have the proper permissions.

Spend some time with the proc(1) man page and familiarize yourself with the commands detailed therein. You'll find ways to list the libraries used by a process, the signal disposition for a process, and the process's credentials--you can even stop and restart processes!

---

## Writing /proc tools:

In the /proc file system, the address space of another process can be accessed with read and write system calls, which allows the debugger to access a process being debugged with much greater efficiency. The page 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.

The real beauty of /proc is that anything you can possibly want to know about a process is there; you just ask for it. Two structures, prstatus and prpsinfo, both defined in /usr/include/sys/procfs.h, can be populated with a wealth of information about a process.

Here's one example. A developer wants to know the total image size, the resident set size, the heap size, and the stack size. Further, he wishes to track this data over time in a manner similar to vmstat(1M). All told, a seemingly daunting task!

However, we can simplify this programmatic challenge by using the /proc file system. The tool called memlook will display memory statistics for a given PID. In addition, you can give memlook an interval, which will cause it to recheck the memory usage at a specified interval of seconds. This capability is particularly handy for trend analysis. Here's a sample output from memlook.

```
vlsi> ps
PID TT    S TIME COMMAND
245 pts/9  S  0:00 -csh
vlsi> memlook 245
PID  IMAGE    RSS      HEAP      STACK
245  1499136  1044480  24581    8192
```

**The source code to memlook.**

```
/*Lab1.c */
/*  memlook.c -- A process memory utilization reporting tool.
*/

#include < stdio.h >
#include < sys/types.h >
#include < sys/signal.h >
#include < sys/syscall.h >
#include < sys/procfs.h >
```



```

#include < sys/param.h >
#include < fcntl.h >

int counter = 10;

int showUsage(const char *);
void getInfo(int, int);

int main (int argc, char *argv[])
{
    int fd, pid, timeloop = 0;
    char pidpath[BUFSIZ];

    switch (argc) {
        case 2:
            break;
        case 3:
            timeloop = atoi(argv[2]);
            break;
        default:
            showUsage(argv[0]);
            break;
    }

    pid = atoi(argv[1]);
    sprintf(pidpath, "/proc/%-d", pid);

    if ((fd = open(pidpath, O_RDONLY)) < 0) {
        perror(pidpath);
        exit(1);
    }

    if (0 < timeloop) {
        for (;;) {
            getInfo(fd, pid);
            sleep(timeloop);
        }
    }

    getInfo(fd, pid);
    close(fd);
    exit(0);
}

int showUsage(const char *progname)
{
    fprintf(stderr, "%s: usage: %s < PID > [time delay]\n"
        , progname, progname);
    exit(3);
}

void getInfo(int fd, int pid)
{

```

```

prpsinfo_t prp;
prstatus_t prs;

if (ioctl(fd, PIOCPSINFO, &prp) < 0) {
    perror("ioctl");
    exit(5);
}

if (ioctl(fd, PIOCSTATUS, &prs) < 0) {
    perror("ioctl");
    exit(7);
}

if (counter > 9) {
    fprintf(stdout, "PID\tIMAGE\t\tRSS\t\tHEAP\t\tSTACK\n");
    counter = 0;
}
fprintf(stdout, "%d\t%-9d\t%-9d\t%-15d\t%-15d\n"
    , pid, prp.pr_bysize, prp.pr_byrssize
    , prs.pr_brksize, prs.pr_stksize);
counter++;
}

```

---

## Exercises

### Note:

Lab Problems will be given during the lab based on material covered in this lab manual