by S. Lee Henry

KEN CALL

The /proc File System

The `/proc` file system is an example of the `procfs` file system type that was new to SunOS with Solaris 2.x. For systems administrators who seldom wander into the `/proc` directory, its utility may seem practically nonexistent. What does one need, after all, with images of running processes? What could one *do* with them? Good questions.

The concept of a *virtual* file system seems odd at first and no more appealing than a virtual lunch. Try watching the faces of fellow systems dweebs when technotalk delves into virtual file systems and you'll see what I mean.

"Virtual file systems," I grin. "They don't take up disk space." "Good idea," says the guy watching the coffee pot filling up. "Maybe they should *all* be virtual. Think of the disk space I'd save."

In fact, `procfs` file systems (only one of a number of virtual file system types), don't occupy space on disk, but look and act like disk-based file systems. What I mean by this is that they provide us with a familiar interface for debugging running processes and add to the information available to us through commands like `ps`.

Take a look at the files in `/proc`. You'll notice that their names reflect the `procIDs` of the particular processes with prepended zeroes (for example, `/proc/00441`). Each file is owned by the user running the particular process, and only he has read and write privileges. The size of each file is the size of the process image. Try the command in Figure 2. Interestingly, all of the files are in units of 4 KB. This isn't a coincidence, of course, but tells us something about the way memory is allocated. If you watch these files over time, some of them will disappear. The commands or applications that they correspond to will finish processing or be killed. Others will appear to be constant. Take a look at `/proc/00000`, for example. This file has a size of 0 and a date reflecting your last reboot. This file corresponds to the scheduler. The `/proc/00001` file corresponds to the `init` process. You'll notice that this file also has the timestamp of your most recent reboot (even if you've changed run states since), as shown in Figure 4. Notice that the size of `/proc/00001` is considerably larger than the size of the `/usr/sbin/init` binary. Executing processes allocate memory for data manipulation and storage, so their process images are always larger than the binaries on disk (see Figure 1).

Most process images will stay the same size while they are running. That is, they allocate the memory required up front. Processes can change size, especially when they suffer from memory leaks.

Memory leaks occur when memory

Figure 1. Process Image Size

```
-r-xr-xr-x 1 root   sys    28064 Jul 16 1994 /usr/sbin/init
```

Figure 2. /proc File Sizes

```
radman:/home/svr=> foreach SIZE (`ls -l /proc | awk '{print $5}'`)
? echo $SIZE | awk '{print $1 / 4096}'
? end
0
197
0
0
2004
2008
2008
2004
1270
437
328
364
350
338
435
```

Figure 3. The /proc/00001 File

```
boson:/home/slee=>    ls -l   /proc/00000  /proc/00001  /proc/00441
-rw-------    1 root      root          0  Sep 16  17:31  /proc/00000
-rw-------    1 root      root     806912  Sep 16  17:31  /proc/00001
-rw-------    1 slee      dweebs   954368  Sep 16  17:34  /proc/00441
```

Figure 4. The ps Command

```
boson:/home/slee=> ps -efl | head -12
 F S  UID PID PPID   C   PRI  NI      ADDR   SZ      WCHAN     STIME   TTY    TIME   COMD
19 T root   0    0  66     0  SY  f0187950    0             17:31:52   ?      0:01   sched
 8 S root   1    0  80    41  20  fc0bd998  187   fc0bdb68  17:31:55   ?      0:05   /etc/init
 8 S slee 441  438  36    99  20  fc41d990  233   fc52d71e    Sep 16   pts/8  0:00   /bin/csh
```

Figure 5. Sample C Program

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/signal.h>
#include <sys/fault.h>
#include <sys/syscall.h>
#include <sys/procfs.h>
#include <stdio.h>
#include <netinet/in.h>
#include <errno.h>
#include <fcntl.h>
#if 0
typedef struct prstatus {
        long    pr_flags;                 /* Flags (see below) */
        short   pr_why;                   /* Reason for process stop (if stopped) */
        short   pr_what;                  /* More detailed reason */
        siginfo_t pr_info;                /* Info associated with signal or fault */
        short   pr_cursig;                /* Current signal */
        u_short  pr_nlwp;                 /* Number of lwps in the process */
        sigset_t pr_sigpend;              /* Set of signals pending to the process */
        sigset_t pr_sighold;              /* Set of signals held (blocked) by the lwp */
        struct  sigaltstack pr_altstack;  /* Alternate signal stack info */
        struct  sigaction pr_action;      /* Signal action for current signal */
        pid_t   pr_pid;                   /* Process id */
        pid_t   pr_ppid;                  /* Parent process id */
        pid_t   pr_pgrp;                    /* Process group id */
        pid_t   pr_sid;                     /* Session id */
        timestruc_t pr_utime;               /* Process user cpu time */
        timestruc_t pr_stime;               /* Process system cpu time */
        timestruc_t pr_cutime;              /* Sum of children's user times */
        timestruc_t pr_cstime;              /* Sum of children's system times */
        char    pr_clname[PRCLSZ];          /* Scheduling class name */
        short   pr_syscall;                 /* System call number (if in syscall) */
        short   pr_nsysarg;                 /* Number of arguments to this syscall */
        long    pr_sysarg[PRSYSARGS];       /* Arguments to this syscall */
        id_t    pr_who;                     /* Specific lwp identifier */
        sigset_t pr_lwppend;                /* Set of signals pending to the lwp */
        struct ucontext *pr_oldcontext;     /* Address of previous ucontext */
        caddr_t  pr_brkbase;                /* Address of the process heap */
        u_long pr_brksize;                  /* Size of the process heap, in bytes */
        caddr_t  pr_stkbase;                /* Address of the process stack */
        u_long pr_stksize;                  /* Size of the process stack, in bytes */
        short   pr_processor;               /* processor which last ran this LWP */
        short   pr_bind;                    /* processor LWP bound to or PBIND_NONE */
        long    pr_instr;                   /* Current instruction */
        prgregset_t pr_reg;                 /* General registers */
} prstatus_t;
#endif

char    *progname;
char    *procfnum;
FILE    *procfile;
int     procfd;
int     retval;
struct  stat *procstatus;

void
main ( int argc, char **argv )
{
char ch=' ';

struct prstatus p;

progname = *argv++;
argc--;

if(argc == 0) {
   printf("Usage: %s <filename>\n",progname);
   return;
}

procfnum = *argv;

if ((procfd = open(procfnum,O_RDONLY)) == NULL) {
   printf("cannot open input file\n");
   return;
}

if (retval = ioctl(procfd, PIOCSTATUS, &p) == BADRET) {
   printf("unable to access file %s\n",procfnum);
   printf("errno = %i\n",errno);
   return;
}

printf("parent process is:      %i\n",p.pr_ppid);
printf("size of process heap:   %i\n",p.pr_brksize);
printf("size of process stack:  %i\n",p.pr_stksize);
}
```

allocated by a process is not deallocated when it is no longer needed. Available memory appears to dwindle. Several memory leaks can cause a process to run out of memory and can chew up the swap space available on your system to a point at which all processes begin to suffer.

The “beauty” of the `/proc` file system is that it provides a way to query and, with care, control running processes without requiring that the processes be child processes of a debugger. Using standard system calls–`open(2)`, `close(2)`, `read(2)`, `write(2)` and `ioctl(2)`–you can manipulate the process images in much the same way as you would any standard file (any standard *binary* file, that is). Remember, these files will resemble core dumps more closely than they will resemble letters to Mom.

To make this point a little clearer, I’ve included a sample C program (see Figure 5). This program pulls some data out of whatever `/proc` file is given as an argument and displays it. Notice that the data structure used is separated by `#if 0` and `#endif` commands so that it is not compiled with the rest of code. This structure is defined in the `procfs.h` header file and is included with the code only to make it easier to follow.

Much of the information available with the `/proc` file system is also available through the `ps` command, as shown in Figure 4.

Take a look at the `proc` man page for more information on the data available through the `procfs` interface.

***S. Lee Henry** is a security engineer at Infonet in El Segundo, CA, and lives in a part of LA that may secede and be called Gridlockia before you read this column. Send your virtual thoughts to her using the address* `slee@cpg.com`.