Up until now we've talked about mostly static things: properties of files and users. Now we'll start looking at dynamic things: the programs that are actually running on the computer.

The login shell provides us with some tools for managing the programs we run. We've already talked about some of these, like pipes and redirects. We'll expand on these today, and add some new tools.

Then we'll talk about the underlying processes that actually do the work, and see how to control those, beyond what the shell can do for us.

Finally, we'll introduce one special process, called “init”.

But first, we'll talk a little more about the good old days...
Part 1: Terminals

Long ago, people communicated with computers through terminals, like the one on the left, which were connected to the computer by serial communication lines. Bits sent back and forth through these serial lines represented characters typed on the keyboard, or text to display on the monitor. Although few people use serial terminals any more, the basic mechanisms of communicating through a terminal persist. Now we use “terminal emulator” programs, like xterm or gnome-terminal, to communicate with the computer when we're running the X window system. Each of these terminal emulators behaves just like the old serial terminals. Whenever we talk to a command line, we're still communicating through a terminal (or “pseudo-terminal”) interface, just as we did long ago.

Because of this, it's useful to understand how information is passed to and from terminals.
Character Encoding:

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>U</td>
</tr>
<tr>
<td>B</td>
<td>V</td>
</tr>
<tr>
<td>C</td>
<td>W</td>
</tr>
<tr>
<td>D</td>
<td>X</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
</tr>
<tr>
<td>F</td>
<td>Z</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>2</td>
</tr>
<tr>
<td>I</td>
<td>3</td>
</tr>
<tr>
<td>J</td>
<td>4</td>
</tr>
<tr>
<td>K</td>
<td>5</td>
</tr>
<tr>
<td>L</td>
<td>6</td>
</tr>
<tr>
<td>M</td>
<td>7</td>
</tr>
<tr>
<td>N</td>
<td>8</td>
</tr>
<tr>
<td>O</td>
<td>9</td>
</tr>
<tr>
<td>P</td>
<td>10</td>
</tr>
<tr>
<td>Q</td>
<td>11</td>
</tr>
<tr>
<td>R</td>
<td>12</td>
</tr>
<tr>
<td>S</td>
<td>13</td>
</tr>
<tr>
<td>T</td>
<td>14</td>
</tr>
</tbody>
</table>

Prior to the 1960s, the most widespread way of communicating data electronically was morse code. When a telegram was sent, its text was encoded in morse code and transmitted through air or a wire to its destination, where it was decoded back into text.

Morse code was fine for human telegraphers, but it was clumsy for computers. In the 1960s the “American Standards Association” published a new, more computer-friendly way of transmitting text. This was called the American Standard Code for Information Interchange (ASCII).

In ASCII, each character is represented by 8 bits of information (1 byte). When you store text in a file on disk, the text is stored as ASCII characters. ASCII characters are also the way communications between a terminal (or pseudo-terminal) and a computer are encoded.

(Actually, other encodings like UTF-8 may be used these days, but the principle is the same. For simplicity, let’s just assume everything is ASCII.)
Displaying Text on a Terminal:

```
~/demo> cat test.txt
01010100010010000100100101010011001000000100100101010011
00100000010000010010000001010100010001010101001101010100
```

If you type a command like “cat test.txt” at the command line, and test.txt is a text file containing the string “THIS IS A TEST”, this is what happens. The data in the file is stored on disk as a string of binary ASCII characters. This data is sent to your terminal, which displays it by decoding the ASCII data back into characters.

Old-fashioned serial terminals typically displayed characters on a grid 80 characters wide and 24 characters tall. This is still the default size for most terminal emulator programs.

Also note that when you typed “cat test.txt”, the terminal emulator converted your command into ASCII and transmitted it to your shell.
Information about Your Terminal:

What's the name of my terminal?

```bash
~/demo> tty
/dev/pts/5
```

A "pseudo-terminal" created for a terminal emulator window like xterm or gnome-terminal running under the X window system, or a remote login via ssh.

```bash
~/demo> tty
/dev/tty1
```

A "virtual console". You'll see this if you sit down at a Linux computer that's not running X.

What type of terminal is it?

```bash
~/demo> echo $TERM
xterm
```

This says that the terminal is either the "xterm" terminal emulator, or some other program that acts like an xterm.

```bash
~/demo> echo $TERM
linux
```

This is what you'll see when sitting at a Linux computer that's not running X.
The “who” and “w” Commands:

The “who” command tells you who's logged in interactively, which terminals they're using, and when they logged in. (There's also some information about X displays, but we'll save that for later.)

```
~/demo> who
elvis pts/8  2009-02-04 07:28 (:1001)
elvis pts/12 2009-01-29 07:30 (:1000)
```

The “w” command gives you more information. It also tells you about each user's idle time and CPU usage, and tells you what program the user is currently running.

```
~/demo> w
  10:28:02 up 40 days, 5 min,  3 users, load average: 0.18, 0.20, 0.12
 USER    TTY      FROM     LOGIN@    IDLE    JCPU   PCPU WHAT
elvis    pts/8    :1001    07:28     2:50m   2.18s  0.00s -bin/tcsh
elvis    pts/12   :1000    29Jan09   6days   2.20s  0.10s -bin/tcsh
```

Note that both “who” and “w” may display inaccurate or misleading information about users who are logged in locally through terminal emulator windows under X. The information should be accurate for remote logins through ssh, though.
The bash and tcsh command line shells provide users with some tools to manage multiple simultaneous jobs.

Jobs are a convenient way of managing the underlying processes that are really doing the work. There are many mechanisms for starting and managing processes. The shell's job control mechanism is just one of them.

In this section, we'll talk about how the shell handles jobs, and then we'll move on to talking about the underlying processes in the next section.
Canceling a Command with Ctrl-C:

```bash
~/demo> myprogram
Processing number 0 ...
Processing number 0 ...
Processing number 0 ...
Processing number 0 ...
Processing number 0 ...
Processing number 0 ...
Processing number 0 ...
```

Most interactive command-line programs can be killed by typing **Control-C** (that is, holding down the “ctrl” key and pressing “c”). This sends a signal to the running process, telling it to terminate immediately. The process has the option of ignoring this signal, but most programs will honor it.

Oh no! I forgot to tell it to increment the count! It's in an infinite loop!
Suspending/Resuming with Ctrl-Z and “fg”:

```
~/demo> myprogram
Processing number 0 ...
Processing number 1 ...
Processing number 2 ...
Processing number 3 ...
Processing number 4 ...
Processing number 5 ...
Processing number 6 ...

Suspended
```

```
~/demo> jobs
[1] + Suspended myprogram
```

```
~/demo> fg
myprogram
Processing number 7 ...
Processing number 8 ...
Processing number 9 ...
Processing number 10 ...
```

You can suspend a running program by typing Ctrl-Z. The program will stop executing, but it will remain frozen in memory. You start it again with the “fg” command (for “foreground”). The “jobs” command will show you suspended jobs.

The number in square brackets at the beginning of the output is a “job specifier”. If you have more than one job, you can tell fg which one to foreground by giving it the job specifier, preceded by a “%” sign. (E.g., “fg %1”).

When you log out, you'll be warned if you have any suspended jobs. If you continue to log out, the jobs will die.
The “jobs” Command:

Here's an example showing the output of the “jobs” command when the shell is managing several jobs:

```
~/demo> jobs
[1]  + Suspended     myprogram
[2]  - Suspended     myprog2
[3]    Suspended     otherprog
[4]    Suspended     prog4
```

The columns are:

- **Job Specifier.** This is a number that uniquely identifies the job. In commands like “fg”, you can choose a particular job by giving a “%” and the job specifier, like “fg %2”.

- **Current.** A “+” in this field means that this is the “current” job. Commands like “fg”, if not explicitly given a job specifier, will operate on this job. A “-” in this field means that this is the “next” job. It will become the current job when the current job finishes.

- **State.** This can be “Suspended”, “Running”, “Stopped” or “Done”.

- **Command.** The name of the command being run by this job.
Sending Jobs to the Background with “bg”:

```bash
$ demo> myprogram
Processing number 0 ...
Processing number 1 ...

Suspended
$ demo> jobs
[1] + Suspended

$ demo> bg
[1] myprogram &
We get a command prompt back

$ demo> ...
Processing number 7 ...
Processing number 8 ...
Processing number 9 ...
Processing number 10 ...
```

But any output will continue to pop up, whenever the program says anything.

If you want a job to continue running while you go on and do other things, you can use the “bg” (for “background”) command to put it into the background. The job will continue to run, and you’ll see any output it generates.

This is useful, but it has two problems:

1. The output from your program may be annoying while you're trying to do other things.

2. If you log out, your backgrounded jobs may die. This will happen if the backgrounded job tries to read or write anything interactively. Once you've logged out, the program will get an error if it tries to ask you for input or write any output to the terminal, because those devices are no longer available.
Redirecting stdin/stdout/stderr for Background Jobs:

By default, stdin is connected to the terminal's keyboard, and stdout and stderr are connected to the terminal's display...

...but we can redirect them elsewhere, to eliminate dependence on interactive I/O devices:

Tcsh syntax:  myprogram < file.in >& file.out
Bash syntax:  myprogram < file.in > file.out 2>&1

If we can connect the I/O channels to something other than our current display and keyboard, we can think about putting the job in the background, logging out, and coming back later to check on it.

We've already seen how to redirect stdout and stderr. Now we see that stdin can be redirected also, with the “<” character.

The commands above say “run myprogram, reading input from 'file.in' and writing output and errors into 'file.out'.”

We could start the program this way, and then type Ctrl-Z to suspend it, and “bg” to background it, or....
Starting a Job in the Background:

Tcsh syntax:
```bash
~/$demo> myprogram < /dev/null >& myprogram.out &
```

Bash syntax:
```bash
~/$demo> myprogram < /dev/null > myprogram.out 2>&1 &
```

Appending an ampersand to the end of a command causes the command to be put into the background immediately.

```bash
~/$demo> jobs
[1] + Running ./myprogram < /dev/null >& myprogram.out
```

If you don't need to give your program any input, and if you don't want to create an empty file to point stdin at, you can just use the handy “null device”, `/dev/null`. This should always be present.

By default, when the shell starts a program the program's stdin, stdout and stderr all point to the user's terminal. Remember the general principle in Linux that “everything is a file”? Well, the terminal appears to the operating system as though it were just another file, with a name like “/dev/pts/3” or some such. By default, the shell points stdin, stdout and stderr to this file, just as though you'd typed something like:

```bash
myprogram < /dev/pts/3 >& /dev/pts3
```
The shell's job control features just provide a convenient way of dealing with processes running in the operating system.

The shell's fg and bg commands, the jobs command, and all the tools for controlling jobs, are just handles to make the management of the underlying processes more convenient. You might think of the relationship between “job” and “process” as something like the relationship between bowling-ball-case and bowling ball.

Jobs are just containers for processes spawned off by the shell. The shell gives you a convenient “job specifier” to refer to each job, separate from the “process ID” of the contained process. The shell can connect jobs to your terminal, or to each other, through the processes' stdin and stdout channels.
The Shell creates pipelines by creating jobs and connecting the jobs' processes together through their stdin and stdout channels.
Notice that each shell has its own set of “job specifiers”, but the “process IDs” (PIIDs) are unique across the whole operating system.
Part 3: Processes

So, what are processes?
Every process has a unique numerical identifier called a “process ID” or “PID”, analogous to the UID for a user or the GID for a group.

The ownership of a process is specified by UID and GID values stored within the process. Each process has two pairs of these: “effective” UID and GID and “real” UID and GID.

Each process has an associated “executable”, which is usually a program stored on disk. (Sometimes the kernel will create processes on its own, and these don't point to a separate program.)

The process has a list of file descriptors, which point to the process's stdin/stdout/stderr and any other files the program currently has open for reading or writing.
The “ps” Command:

The “ps” command can show information about processes, much as the “ls” command shows information about files.

The “user” field shows the username of the owner of each process.

%CPU and %Mem show what fraction of these resources the process is currently using.

VSZ and RSS are two measures of how much memory the process is using.

The TTY column shows the name of any terminal that's attached to this process.

The STAT column tells us what the process is currently doing. Entries beginning with “R” are running. Entries beginning with “S” are sleeping.
The “top” command:

The "top" command gives a continuously-updating display showing what’s happening on a computer. It packs a lot of information into a small space. The upper part of the display has information about users, memory, load, etc., and the lower part of the display has information about processes. By default, the display is sorted so that processes using the largest fraction of the available CPU time are displayed at the top.

```bash
$ demo> top
top - 18:23:14 up 187 days,  0:08,  4 users,  load average: 1.05, 1.03, 1.00
Tasks: 215 total,  2 running, 213 sleeping,  0 stopped,  0 zombie
Cpu(s):  0.0%/us,  0.2%/sy,  0.0%/ni,  99.0%/id,  0.1%/wa,  0.0%/hi,  0.0%/si,  0.0%/st
Mem:   4017152k total, 3990516k used,  26636k free,  3776k buffers
Swap:  2031608k total,  333196k used, 1698412k free,  3511864k cached

PID USER   PR NI  VIRT  RES  SHR %CPU %MEM    TIME+  COMMAND
32043 ef2p  25   0   7075 34m  14m 100  0.9 4:24:05 analyzer
32197 blkdev 15   0  2436 928  660  2 0.0 0:00.01 top
  1 root    15  0  2000  632  544  S  0.0 0:01.60 init
   2 root    15  0   0   0    0    0  S  0.0 0:19.57 migration/0
   3 root    34  19   0   0    0    0  0.0 0:14.26 ksoftirqd/0
   4 root    15  0   0   0    0    0  S  0.0 0:00.00 watchdog/0
   5 root    15  0   0   0    0    0  S  0.0 0:12.85 migration/1
   6 root    34  19   0   0    0    0  0.0 0:04.38 ksoftirqd/1
   7 root    15  0   0   0    0    0  S  0.0 0:00.00 watchdog/1
   8 root    15  0   0   0    0    0  S  0.0 0:00.00 watchdog/1
   9 root    39  19   0   0    0    0  0.0 0:03.24 ksoftirqd/2
  10 root    15  0   0   0    0    0  S  0.0 0:00.00 watchdog/2
  11 root    15  0   0   0    0    0  S  0.0 0:11.41 migration/3
  12 root    34  19   0   0    0    0  0.0 0:03.63 ksoftirqd/3
```

Some interesting information:

- **Uptime.** This shows how long the computer has been running.

- **Number of users currently logged in.**

- **Load average.** Think of this as the average number of processes running at any given time. The three numbers are rolling averages over 1, 5 and 15 minutes.

- **Memory.** This shows the amount of total memory and free memory, and has data about swap space usage.
The “watch” Command:

Wouldn’t it be nice if you could make any command continuously-updating, like top? You can, with the “watch” command. By default, watch re-executes a command every two seconds, and updates the display to show how the command’s output has changed.

~/demo> watch w

Every 2.0s:  w

10:59:20 up 40 days, 35 min, 2 users,  load average:  0.06, 0.17, 0.13
USER     TTY      FROM    LOGIN@     IDLE    JCPU   PCPU   WHAT
elvis    pts/12   :1000   29Jan09   6days  2.21s  0.10s   -bin/tcsh
elvis    pts/8    :1001   07:28      2:59m  2.18s  0.09s   -bin/tcsh

~/demo> watch stat junk.dat

Every 2.0s:  stat junk.dat

File:  `junk.dat`
Size:  14 Blocks:   8 I/O Block: 4096 regular file
Device: fd00h/64768d Inode: 10321951 Links:   1
Access: (0644/-rwxr-xr-x) Uid: ( 500/    bkw1a) Gid: ( 501/    bkw1a)
Modify: 2009-02-04 09:28:04.000000000 -0500
Change: 2009-02-04 09:28:04.000000000 -0500
Niceness:

One of the columns “top” displays is “NI”, for “Niceness”. Each process has a niceness value between -20 and 20. By default, processes start off with a niceness of zero. Processes with higher niceness are more willing to give up the CPU when another process wants to use it. Critical system processes are often run at negative niceness values, since it’s important that they have access to the CPU when they need it. Low-priority processes may run with a very high niceness, so they only get the CPU when nothing else wants it. Non-root users can adjust the niceness of their processes, but they can’t lower it below its initial value.

```bash
~$ demo> renice 15 32043 # (assuming I'm ef2p!)
```

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>32043</td>
<td>ef2p</td>
<td>25</td>
<td>15</td>
<td>70756</td>
<td>34m</td>
<td>14m</td>
<td>R</td>
<td>100</td>
<td>0.9</td>
<td>4:24.05</td>
</tr>
<tr>
<td>32197</td>
<td>bkw1a</td>
<td>15</td>
<td>0</td>
<td>2436</td>
<td>928</td>
<td>660</td>
<td>R</td>
<td>2</td>
<td>0.0</td>
<td>0:00:01</td>
</tr>
<tr>
<td>1 root</td>
<td>15</td>
<td>0</td>
<td>2060</td>
<td>632</td>
<td>544</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>0:01:00</td>
<td>init</td>
</tr>
<tr>
<td>2 root</td>
<td>RT</td>
<td>-5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>0:19:57</td>
<td>migration/0</td>
</tr>
<tr>
<td>3 root</td>
<td>34</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>0:14:26</td>
<td>ksoftirqd/0</td>
</tr>
<tr>
<td>4 root</td>
<td>RT</td>
<td>-5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>0:00:00</td>
<td>watchdog/0</td>
</tr>
<tr>
<td>5 root</td>
<td>RT</td>
<td>-5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>0:12:85</td>
<td>migration/1</td>
</tr>
<tr>
<td>6 root</td>
<td>34</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>0:04:28</td>
<td>ksoftirqd/1</td>
</tr>
<tr>
<td>7 root</td>
<td>RT</td>
<td>-5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>0:00:00</td>
<td>watchdog/1</td>
</tr>
<tr>
<td>8 root</td>
<td>RT</td>
<td>-5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>0:09:87</td>
<td>migration/2</td>
</tr>
<tr>
<td>9 root</td>
<td>39</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>0:03:24</td>
<td>ksoftirqd/2</td>
</tr>
<tr>
<td>10 root</td>
<td>RT</td>
<td>-5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>0:06:08</td>
<td>watchdog/2</td>
</tr>
<tr>
<td>11 root</td>
<td>RT</td>
<td>-5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>0:11:41</td>
<td>migration/3</td>
</tr>
<tr>
<td>12 root</td>
<td>34</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>0:03:63</td>
<td>ksoftirqd/3</td>
</tr>
</tbody>
</table>
The “kill” and “killall” Commands:

How can we control a process? One way is by sending it “signals”.

The “kill” command can be used to send signals to processes. The name is misleading, because only some of the signals will normally result in killing the process. Killing processes was the original purpose of the command, and the name stuck even after the command's purpose was expanded. (Maybe a better name would be “signal”.)

By default, the “kill” command will send a “SIGTERM” signal to a process. You can modify this behavior by specifying a signal, either by number or by name.
The SIGTTIN and SIGTTOU signals are received by a process when it tries unsuccessfully to read input or write output to a terminal. By default, these signals cause the process to stop, just as a SIGSTOP would do.

References like “alarm(2)” and “abort(3)” refer to man pages. For example, “alarm(2)” refers to the man page for the term “alarm” in section 2 of the man pages. You could view it by typing “man 2 alarm”.
Signal Actions:

Several pre-defined actions are available for processes to take when they receive a signal. Processes may also define their own signal-handling functions for any signals except SIGKILL and SIGSTOP.

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
<td>Terminate the process.</td>
</tr>
<tr>
<td>Ign</td>
<td>Ignore the signal.</td>
</tr>
<tr>
<td>Core</td>
<td>Terminate the process and dump core (see core(5)).</td>
</tr>
<tr>
<td>Stop</td>
<td>Stop the process.</td>
</tr>
<tr>
<td>Cont</td>
<td>Continue the process if it is currently stopped.</td>
</tr>
<tr>
<td>Catch</td>
<td>Call a predefined signal-handling function.</td>
</tr>
</tbody>
</table>
The `/proc Filesystem:

The `/proc directory doesn't exist on-disk. It's created in memory and kept up-to-date by the kernel. If you look into `/proc, you'll see a bunch of directories, each with a number as its name. Each of these numbers is the PID of a process, and the directory contains information about the process with that PID. Here's a typical example:

```
[root@demo ~]# ls -l /proc/19335
total 0
-r-xr-xr-x 2 elvis elvis 0 Feb 3 15:13 attr
-r-------- 1 elvis elvis 0 Feb 3 15:13 auxv
-r--r--r-- 1 elvis elvis 0 Feb 3 15:13 cmdline
-r-------- 1 elvis elvis 0 Feb 3 15:13 coredump_filter
-r-------- 1 elvis elvis 0 Feb 3 15:13 cpuset
lrwxrwxrwx 1 elvis elvis 0 Feb 3 15:13 cwd -> /home/elvis
lrwxrwxrwx 1 elvis elvis 0 Feb 3 15:13 environ
lrwxrwxrwx 1 elvis elvis 0 Feb 3 15:13 exe -> /usr/bin/top
dr-x------ 2 elvis elvis 0 Feb 3 15:13 fd
-r-------- 1 elvis elvis 0 Feb 3 15:13 limits
-r-------- 1 elvis elvis 0 Feb 3 15:13 loginuid
-r-------- 1 elvis elvis 0 Feb 3 15:13 maps
-r-------- 1 elvis elvis 0 Feb 3 15:13 mem
-r-------- 1 elvis elvis 0 Feb 3 15:13 mounts
-r-------- 1 elvis elvis 0 Feb 3 15:13 mountstats
-r-------- 1 elvis elvis 0 Feb 3 15:13 oom_adj
-r-------- 1 elvis elvis 0 Feb 3 15:13 oom_score
-r-------- 1 elvis elvis 0 Feb 3 15:13 schedstat
-r-------- 1 elvis elvis 0 Feb 3 15:13 smaps
-r-------- 1 elvis elvis 0 Feb 3 15:13 stat
-r-------- 1 elvis elvis 0 Feb 3 15:13 statm
-r-------- 1 elvis elvis 0 Feb 3 15:13 status
dr-xr-xr-x 3 elvis elvis 0 Feb 3 15:13 task
-r--r--r-- 1 elvis elvis 0 Feb 3 15:13 wchan
```

The `/proc filesystem contains everything you should need to know about processes running on the computer. Tools like “ps” and “top” look at `/proc to get the data they display for you. You can also look at `/proc directly if you want. Each file in `/proc contains only plain text.
The “lsf” Command:

The lsf command can list the open files associated with a given process.

```
[root@demo ~]# lsf -p 19335
COMMAND   PID USER  ED  TYPE DEVICE SIZE NODE NAME
--- ------ --- --- ----- ------ ---- ----
top 19335 elvis   cd  DIR 253,0  4096 29592275 /home/elvis
top 19335 elvis   rtd  DIR 253,0  4096 2 /
top 19335 elvis   txt  REG 253,0  62200 33185251 /usr/bin/top
top 19335 elvis   mem  REG 253,0  46880 34766935 /lib/libnss_files-2.5.so
top 19335 elvis   mem  REG 253,0 15032 34768080 /lib/libncurses.so.5.6
top 19335 elvis   mem  REG 253,0 125736 34766911 /lib/ld-2.5.so
top 19335 elvis   mem  REG 253,0 164588 34766856 /lib/i686/nosegneg/libc-2.5.so
top 19335 elvis   mem  REG 253,0  53856 34768103 /lib/libproc-3.2.7.so
top 19335 elvis   mem  REG 253,0  6428  34766870 /lib/libdl-2.5.so
top 19335 elvis   mem  REG 253,0  65860 34766879 /lib/libinfo.so.5.6
```

The lsf command also gets its information from /proc.
The “fuser” Command:

The fuser command is like the inverse of lsof. It tells you what processes are using a given file or directory.

```
[root@demo ~]# fuser /usr/bin/top
/usr/bin/top: 19335e
```

```
[root@demo ~]# fuser /lib/libdev-2.5.so
/lib/libdev-2.5.so:  1m 1275m 1345m 1347m 2153m 2167m 4422m
4441m 4446m 8500m 8512m 12605m 12615m 12623m 12626m 13950m 13968m
14735m 14766m 15742m 15744m 18110m 18163m 18165m 18166m 18280m 18281m
18283m 18288m 18290m 18314m 18569m 18572m 18573m 18588m 18590m 18592m
18822m 18823m 18825m 18828m 18829m 18872m 18874m 18877m 18878m 18926m
18929m 18931m 18933m 18940m 18941m 18943m 18945m 18946m 18952m 18960m
18974m 19001m 19016m 19303m 19335m 20306m 20307m 20308m 20309m 20348m
23073m 23547m 26132m 26134m 26183m 26185m 26190m 26192m 26492m 26494m
26724m 26725m 26726m 26730m 26738m 26774m 26776m 26779m 26780m 26823m
26828m 26831m 26833m 26835m 26843m 26844m 26846m 26848m 26849m 26866m
26865m 27449m 27451m 27943m 27944m 28330m 28334m 28385m 28558m 29746m
29983m 30920m 31212m 31214m
```

Fuser, too, gets its information from /proc. The letters after the PIDs tell you how the file is being used by the given process. The “e” in the first example means that the process is executing this file.
The “strace” Command:

The strace command allows you to attach to a running process and see the system functions it calls as it works. This makes strace a very useful debugging tool.

[root@demo ~]# strace -f -p 12345
ioctl(0, FIONREAD, [134917803]) = -1 ENOTTY (Inappropriate ioctl for device)
read(0, "", 1) = 0
ioctl(0, FIONREAD, [134917803]) = -1 ENOTTY (Inappropriate ioctl for device)
read(0, "", 1) = 0
ioctl(0, FIONREAD, [134917803]) = -1 ENOTTY (Inappropriate ioctl for device)
read(0, "", 1) = 0
ioctl(0, FIONREAD, [134917803]) = -1 ENOTTY (Inappropriate ioctl for device)
read(0, "", 1) = 0
ioctl(0, FIONREAD, [134917803]) = -1 ENOTTY (Inappropriate ioctl for device)

The strace output above comes from a problem we recently had with someone's program. It shows that the process was trying to communicate with a terminal on stdin, and was continually calling the “read” and “ioctl” functions on file descriptor zero (stdin).

The program was running detached from a terminal, so the ioctl function failed (“ENOTTY”, means there was an error because there was no terminal available). Reconfiguring the program so that it didn't try to read from stdin fixed the problem.
Part 4: The “init” Process
The “init” Process:

- init is the first process started while a computer is booting.
- It always gets PID=1.
- Init is just a program, usually in /sbin/init. (The boot process has a built-in default value for this path, but it can be overridden manually at boot time.)
- It's init's responsibility to start up all of the other processes to complete the boot process.
- init reads the file /etc/inittab to see what processes need to be started.
- Entries in /etc/inittab are tied to specific "runlevels". The runlevel is a number, usually between 0 and 6, that tells init which set of processes to start.

Note that only a few runlevels have standard meanings across all Linux distributions. Other than these standards, vendors are free to use the runlevels in any way they choose.

We'll talk much more about init when we start talking about network services. For now, this is just a quick introduction.
The /etc/inittab File:

```
# Default runlevel. The runlevels used by RH are:
# 0 - halt (Do NOT set inittdefault to this)
# 1 - Single user mode
# 2 - Multiuser, without NFS (The same as 3, if you do not have networking)
# 3 - Full multiuser mode
# 4 - unused
# 5 - X11
# 6 - reboot (Do NOT set inittdefault to this)
#
# id:5:initdefault:
#
# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

0:wait:/etc/rc.d/rc 0
1:wait:/etc/rc.d/rc 1
2:wait:/etc/rc.d/rc 2
3:wait:/etc/rc.d/rc 3
4:wait:/etc/rc.d/rc 4
5:wait:/etc/rc.d/rc 5
6:wait:/etc/rc.d/rc 6

# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6

# Run xdm in runlevel 5
x:5:respawn:/etc/X11/prefdm -nodaemon
```

Runlevels 0, 1 and 6 (halt, single-user mode and reboot) are standard across all Linux distributions. The use of the other numbers varies widely. The example above is from a Red Hat-derived system.
... but wait....

Lennart Poettering, father of systemd

All of the preceding stuff about "init" was true for many years. Recently, though, many Linux distributions have replaced init with a new tool, "systemd". Systemd aims to replace init and other parts of the operating system with a highly integrated framework that manages booting, network services, and many other things.

Systemd is still highly controversial, and the community of Linux developers is split on whether systemd is a good thing or a bad thing.

We'll talk more about systemd when we address network services and how they're started.
The End

Thanks!