Linux Inspection

This guide has a lot of commands. For each command, it describes the command, shows an example of it running, and describes what the output means. As you walk through it, you should run each command on our own system to see what the output looks like and confirm that the output makes sense for your system.

Part 1: System Status

There are a number of commands that are useful for finding the general status of a system. These can be helpful for debugging why a system is misbehaving (are we out of memory? do we have free disk space?), seeing how loaded a system is (how may people are logged in right now? how many processes are running?), and lots of other basic discovery tasks.

uptime

The `uptime` command gives a super-simple overview of what's going on on the system right now:

```
[root@te-master ~]# uptime
 20:58:59 up  6:56,  1 user,  load average: 0.00, 0.01, 0.05
```

The output of `uptime` is a single line with four fields:

- The current time. In this case, it is almost 9:00pm (`20:58:59`)
- How long the system has been running since last reboot. In this case, the system was powered on six hours and 56 minutes ago (`up  6:56`)
- The number of users that are currently logged in. In this case, there is currently one user logged in. (`1 user`)
- The load on the system. The "load" is, over a given time, the average number of processes that are running or able to run. It is presented as the averages over the last one, five, and 15 minutes (`load average: 0.00, 0.01, 0.05`)

This command is useful for getting a very simple view of the system: how long it's been up, how loaded with people it is, and how loaded with processes it is. If you suspect something is wrong on a system, it can be a place to start narrowing down where to look next.

uname

The `uname` command, short for "Unix name", tells you what operating system kernel is currently running on your system. Running it with the `-a` option gives more information than running the command alone:

```
[root@te-master ~]# uname
Linux
[root@te-master ~]# uname -a
Linux te-master 3.10.0-957.12.2.el7.x86_64 #1 SMP Tue May 14 21:24:32 UTC
2019 x86_64 x86_64 x86_64 GNU/Linux
```

In this case, we're running `Linux` on a system named `te-master`. The release of the Linux kernel being run is `3.10.0-957.12.2.el7.x86_64`, and its version has been set at compile time to have information about when it was compiled: `#1 SMP Tue May 14 21:24:32 UTC 2019`. The name of the machine hardware the kernel is running on is `x86_64` (the first one), its processor is `x86_64` (the second one), and its hardware platform is `x86_64` (the last one). The operating system that the kernel is powering is called `GNU/Linux`.

This command gives an overview of what kind OS and hardware you are dealing with when you log into a system. You may find that a particular system is running a really old kernel, that it is running with an ARM processor, or that you are actually running a different OS than Linux. For example, the output of this same command on MacOS looks like this:

```
Darwin pn1804004.lanl.gov 18.5.0 Darwin Kernel Version 18.5.0: Mon Mar 11
20:40:32 PDT 2019; root:xnu-4903.251.3~3/RELEASE_X86_64 x86_64
```

Note that the MacOS output is in a slightly different format, so you won't be able to directly map the Linux command's fields to this one.

df, du, and free

The `df`, `du`, and `free` commands are used to get information on storage utilization (`df` and `du`) and memory utilization (`free`) on a system. By default, they will each report in bytes or blocks, but they can each take a `-h` option that will make them report their numbers in human-readable format. Looking at them in turn:

df ("disk free")

```
[root@te-master ~]# df -h
Filesystem           Size  Used Avail Use% Mounted on
/dev/sda2            400G  9.9G  390G   3% /
devtmpfs             9.8G     0  9.8G   0% /dev
tmpfs                9.9G     0  9.9G   0% /dev/shm
tmpfs                9.9G  8.6M  9.8G   1% /run
tmpfs                9.9G     0  9.9G   0% /sys/fs/cgroup
172.16.1.253:/data   3.2T   51G  3.2T   2% /data
tmpfs                2.0G     0  2.0G   0% /run/user/0
```

The `df` command lists all of the filesystems on a system, how big they are, what their utilization is, and where they are mounted. In this case, we have lots of space everywhere.

du ("disk used")

```
[root@te-master /]# du -h
93M ./var/lib/rpm
0 ./var/lib/yum/repos/x86_64/$releasever/base
0 ./var/lib/yum/repos/x86_64/$releasever/extras
0 ./var/lib/yum/repos/x86_64/$releasever/updates
```

```
0 ./var/lib/yum/repos/x86_64/$releasever
...
4.0K ./data/ansible/.git/logs/refs/heads
4.0K ./data/ansible/.git/logs/refs
8.0K ./data/ansible/.git/logs
944K ./data/ansible/.git
1.5M ./data/ansible
50G ./data
60G .
```

The du command walks through an entire directory tree (by default, starting in your current directory) and calculates how much storage space is taken up by that directory and its subdirectories. In this case, it ran in the root directory (/), and the output was too long to fully include here. The last line tells us that there is 60GB worth of data in / and all of its subdirectories.

free ("free memory")

```
[root@te-master ~]# free -h
              total        used        free      shared  buff/cache
available
Mem:            19G        596M         18G        8.6M        630M
18G
Swap:            0B          0B          0B
```

The free command tells you how much RAM is available and being used on the system. In this case, the system has a total of 19GB of RAM, of which 596MB are being used. This leaves about 18GB (rounding!) of memory available. The shared field is somewhat misleading: it tells us that 8.6MB is used by RAM-based tmpfs filesystems. You can see that same number in the df output earlier. The buff/cache field is memory used by the kernel for caching, and the available field is similar to the free field, but takes into account the memory being used by the kernel.

These three commands each give insight into the current state of consumable components of your system. They can be used to monitor the usage of these components and take actions if needed: delete files, add more storage, kill processes, add more memory, etc.

w and last

The w and last commands provide information about who is currently or has recently been using your system.

w ("who")

```
[root@te-master ~]# w
 22:50:49 up 2 days,  8:48,  3 users,  load average: 0.00, 0.01, 0.05
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
root     pts/0    te-hyperv        21:31    1.00s  0.11s  0.11s -bash
lowell   pts/2    te-hyperv        22:29   21:29   0.02s  0.02s -bash
cluening pts/3    te-hyperv        22:50    4.00s  0.02s  0.02s -bash
```

The `w` command, which is an abreviation for `who`, starts out by providing the same information that `uptime` provides. It then lists all of the users that are currently logged in to the system, where they are logged in from, when they logged in, how busy they have been, and what they are running.

last ("last users")

```
[root@te-master ~]# last
root     pts/0        te-hyperv        Thu May 30 20:11   still logged in
root     pts/2        te-hyperv        Thu May 30 14:46 - 14:59  (00:13)
root     pts/2        te-hyperv        Thu May 30 14:44 - 14:45  (00:01)
...
root     tty1                          Thu May 23 17:48 - 17:48  (00:00)
reboot   system boot  3.10.0-957.12.2. Thu May 23 17:48 - 18:09  (00:20)
reboot   system boot  3.10.0-957.12.2. Thu May 23 17:40 - 18:09  (00:28)

wtmp begins Thu May 23 17:40:54 2019
```

The `last` command lists all of the logins that have happened on the system since the last time that the `/var/log/wtmp` file was rotated. It includes where they logged in from, when they logged in and out, and how long they were logged in.

These two commands are useful for seeing who is currently using your system and what users' historical usage looks like. `w` is especially useful when you want to reboot a shared system - it shows who is currently logged in, giving you the ability to warn them before they get kicked off.

mount

The mount command is used to attach filesystems to a system, but when it is run without any arguments it will list all of the filesystems that are currently mounted:

```
[root@te-master ~]# mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
devtmpfs on /dev type devtmpfs
(rw,nosuid,size=10268152k,nr_inodes=2567038,mode=755)
securityfs on /sys/kernel/security type securityfs
(rw,nosuid,nodev,noexec,relatime)
...
nfsd on /proc/fs/nfsd type nfsd (rw,relatime)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw,relatime)
/dev/sda2 on / type xfs (rw,relatime,attr2,inode64,noquota)
172.16.1.253:/data on /data type nfs4
(rw,relatime,vers=4.1,rsize=1048576,wsize=1048576,namlen=255,hard,proto=tc
p,timeo=600,retrans=2,sec=sys,clientaddr=172.16.1.254,local_lock=none,addr
=172.16.1.253)
tmpfs on /run/user/0 type tmpfs
```

```
(rw,nosuid,nodev,relatime,size=2055828k,mode=700)
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (rw,relatime)
```

Linux systems use the filesystem abstraction for a lot more than just standard storage, so the output of `mount` may be many lines long. In this example, the two lines that are most immediately interesting are the one that starts with `/dev/sda2` and the one that starts with `172.16.1.253`. The first one is from the second partition on an actual physical disk named `/dev/sdb`. This partition is mounted at the root of the filesystem tree (`/`) and is formatted as an XFS filesystem. The second one is a filesystem that is mounted over the network from `172.16.1.253` and is mounted on the `/data` directory. We don't know the actual format of this filesystem, as it resides on a different system, but we know that it is using the NFS protocol to do the remote mount.

The `mount` command can do a lot more than just list mounts, but running it without any arguments makes it a useful introspection tool for finding what filesystems exist on the system, which ones are local or remote, and what the actual hardware is behind each one.

top ("top processes")

The `top` command is an interactive command that lists all of the processes running on a system, sorting them by by how active they are. By default, it will update once every three or so seconds.

```
[root@te-master ~]# top
top - 22:42:10 up  8:39,  1 user,  load average: 0.00, 0.01, 0.05
Tasks: 235 total,   1 running, 234 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
KiB Mem : 20558264 total, 19299788 free,   612800 used,   645676
buff/cache
KiB Swap:        0 total,        0 free,        0 used. 19502700 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+
COMMAND
 1692 root      20   0       0      0      0 S   0.3  0.0   0:49.82
kworker/3:1
10617 root      20   0  162124   2332   1540 R   0.3  0.0   0:00.08 top
    1 root      20   0  192056   4940   2508 S   0.0  0.0   0:03.74
systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00
kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.00
ksoftirqd/0
    4 root      20   0       0      0      0 S   0.0  0.0   0:00.00
kworker/0:0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00
kworker/0:+
    7 root      rt   0       0      0      0 S   0.0  0.0   0:00.00
migration/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      20   0       0      0      0 S   0.0  0.0   0:01.13
rcu_sched
   10 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 lru-
```

```
add-dr+
   11 root      rt   0       0      0      0 S   0.0  0.0   0:00.14
watchdog/0
```

The first several lines give an overview of the system: the same output as `uptime`, the number of and state of processes on the system, and some statistics on CPU and memory usage. The bottom part of the screen lists individual processes, which by default are sorted by the `%CPU` column. The second line in this section is the `top` command itself, which has process ID `10617` and is being run by `root`. It has a priority of `20`, which is a numeric score used to help the kernel scheduler decide which process to run next. The `NI` column displays this process's "nice" value, which is another value used to help the kernel schedule processes. This process is currently using a total of `162124` bytes of virtual memory on the system, of which `2332` bytes are in physical memory and `1540` bytes are shared with other processes. This process is currently running, as shown by the `R` value in the state (`S`) column, and it is currently using `0.3%` of the system's available CPU time and `0.0%` of its available memory. It has used `0.08` seconds of CPU time since it started.

The `top` command is a very powerful tool for seeing what processes are using the most resources on a system and how that resource usage is changing over time. It is a useful command to leave running in a terminal for an "at a glance" view of the system, and it is also useful for quickly seeing why a system is feels bogged down or if a particular process is making progress.

Part 2: Processes

Processes are one of the base units of execution that exist on a Unix-like system. When you run a program, it starts a new *process* that is assigned a unique *process ID* (PID) number and is named after the name of the program being run. Processes use memory and CPU time, and many of them can be running at once.

ps

We have already seen `top` and how it provides an interactive display of all of the processes running on a system. The `ps` command can be used to get more information about processes on the command line. `ps` has a *very* large number of command line options, and we will explore several of them here.

```
[root@te-master ~]# ps
  PID TTY          TIME CMD
11421 pts/0    00:00:00 bash
11461 pts/0    00:00:00 ps
```

When run with no arguments, `ps` gives a list of processes that are associated with your current login session. In this case, there are two: `bash` (the login shell) and `ps` (which is currently running). Their proccess IDs are `11421` and `11461`, respectively, and neither have used enough CPU time to even register in the `TIME` column.

ps -e

To see all of the processes running on the system, you can give `ps` the `-e` argument to show *every* process. This output will be noticeably longer than the previous output:

```
[root@te-master ~]# ps -e
  PID TTY          TIME CMD
    1 ?        00:00:12 systemd
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
    4 ?        00:00:00 kworker/0:0
    5 ?        00:00:00 kworker/0:0H
...
11407 ?        00:00:00 kworker/7:2
11418 ?        00:00:00 sshd
11421 pts/0    00:00:00 bash
11784 pts/0    00:00:00 ps
21060 ?        00:00:01 kworker/u32:2
```

This gives a better view of the system, but doesn't give an indication of who is running each process.

ps -eF

To get even more information in the `ps` output, you can add the `-F` option to enable *full* output lines.

```
[root@te-master ~]# ps -eF
UID        PID  PPID  C    SZ   RSS PSR STIME TTY          TIME CMD
root         1     0  0 48014  4940   3 May30 ?        00:00:12
/usr/lib/systemd/systemd --switched-root --system --deserialize 22
root         2     0  0     0     0   8 May30 ?        00:00:00 [kthreadd]
root         3     2  0     0     0   0 May30 ?        00:00:00
[ksoftirqd/0]
root         4     2  0     0     0   0 May30 ?        00:00:00
[kworker/0:0]
root         5     2  0     0     0   0 May30 ?        00:00:00
[kworker/0:0H]
...
root     11407     2  0     0     0   7 09:09 ?        00:00:00
[kworker/7:2]
root     11418  4311  0 37118  5448  13 09:10 ?        00:00:00 sshd:
root@pts/0
root     11421 11418  0 28893  2092  10 09:10 pts/0    00:00:00 -bash
root     12037 11421  0 38843  1844  12 09:19 pts/0    00:00:00 ps -eF
root     21060     2  0     0     0   4 02:00 ?        00:00:01
[kworker/u32:2]
```

This output provides a lot more information about the processes, including the process owner (`UID`), the process's parent process (`PPID`), more information about resource utilization, and the complete command line that was used to start the process.

ps -elF

Adding the `-l` argument adds even more information to the line.

```
[root@te-master ~]# ps -elF
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN    RSS PSR STIME TTY
TIME CMD
4 S root         1     0  0  80   0 - 48014 ep_pol  4940   3 May30 ?
00:00:12 /usr/lib/systemd/systemd --switched-root --system --deserialize
22
1 S root         2     0  0  80   0 -     0 kthrea     0   8 May30 ?
00:00:00 [kthreadd]
1 S root         3     2  0  80   0 -     0 smpboo     0   0 May30 ?
00:00:00 [ksoftirqd/0]
1 S root         4     2  0  80   0 -     0 worker     0   0 May30 ?
00:00:00 [kworker/0:0]
1 S root         5     2  0  60 -20 -     0 worker     0   0 May30 ?
00:00:00 [kworker/0:0H]
...
4 D root     11418  4311  0  80   0 - 37118 flush_  5448  13 09:10 ?
00:00:00 sshd: root@pts/0
4 S root     11421 11418  0  80   0 - 28893 do_wai  2092  11 09:10 pts/0
00:00:00 -bash
1 S root     12083     2  0  80   0 -     0 worker     0   7 09:20 ?
00:00:00 [kworker/7:1]
1 S root     12395     2  0  80   0 -     0 worker     0   7 09:25 ?
00:00:00 [kworker/7:2]
4 R root     12940 11421  0  80   0 - 38843 -       1844  14 09:36 pts/0
00:00:00 ps -elF
1 S root     21060     2  0  80   0 -     0 worker     0   4 02:00 ?
00:00:01 [kworker/u32:2]
```

This extra information includes process state (`S`), priority (`PRI`) and nice values (`N`), and the memory size of the process (`SZ`).

ps -elF --forest

All processes have a parent process (except for the `init` process, which is started directly by the kernel). This means that processes can be seen as a tree, or a forest of subtrees rooted on individual processes, when you add the `--forest` argument.

```
[root@te-master ~]# ps -elF --forest
...
4 S root      4311     1  0  80   0 - 28216 poll_s  4252   1 May30 ?
00:00:00 /usr/sbin/sshd -D
4 S root     11418  4311  0  80   0 - 37118 flush_  5448  13 09:10 ?
00:00:00  \_ sshd: root@pts/0
4 S root     11421 11418  0  80   0 - 28893 do_wai  2092  11 09:10 pts/0
00:00:00      \_ -bash
0 R root     12988 11421  0  80   0 - 38919 -       2168  14 09:37 pts/0
00:00:00          \_ ps -elF --forest
...
```

In this example, all of the initial columns are the same as with `-elF`, but the last column includes some ASCII art the indicates the lineage of the processes.

ps All the Things!

The `ps` command is extremely useful and has a lot more options than we have explored here. And the options we looked at here can all be used in different combinations to get different levels of detail. You can read more about the other options available by reading the `ps` manual page: `man ps`.

kill

The `kill` command is used to send signals to processes. Its name comes from its default behavior: to send a signal to a process that will make it terminate.

Trying out `kill` requires opening a second login session so you have two shells running. In one shell, start a `sleep` command that will do nothing for 300 seconds:

```
[root@te-master ~]# sleep 300
```

In a second shell, find that process's PID and use `kill` to terminate it:

```
[root@te-master ~]# ps -e | grep sleep
13801 pts/2    00:00:00 sleep
[root@te-master ~]# kill 13801
```

In your original shell, you should see that the process has been killed:

```
[root@te-master ~]# sleep 300
Terminated
[root@te-master ~]#
```

The `kill` command can be used to terminate processes that aren't running in your login session (so you can't hit `ctrl-C` to kill them). It can also be used to harass your teammates by killing their login shells when they least expect it. But that's not nice, so you definitely shouldn't try it at any point. Even if it would be funny.

Part 3: Special filesystems

As we saw in the `mount` example earlier, Linux (and all Unix-like operating systems) make heavy use of the *file* and *filesystem* abstractions to implement features. Linux has a few special directories that look like standard filesystems, but are actually links into the kernel's memory and configuration. These can be used to get information about data structures within the kernel, and they can be used to change the configuration of a running kernel.

`/proc`

At its base, the `/proc` filesystem provides information about every process that is running on the system:

```
[root@te-master ~]# ls /proc
1/       163/    1933/    32/     4310/  4707/  67/     95/          kpageflags
10/      1692/   1940/    33/     4311/  4709/  68/     956/         loadavg
100/     1693/   1961/    3360/   4312/  4710/  69/     96/          locks
102/     1694/   1962/    34/     4319/  4753/  7/      9613/        mdstat
103/     17/     1963/    35/     4325/  4768/  70/     97/          meminfo
104/     170/    1969/    36/     4333/  48/    71/     98/          misc
...
```

Inside the `/proc` directory is one directory for every process ID on the system, plus some regular files that include information about internal kernel data. Keep in mind that `/proc` is not a real filesystem, and it does not take up space on disk. Instead, each time you run `ls` on the directory or look at a file inside it, the kernel intercepts the underlying system calls and returns the information that corresponds with the command you ran.

The `/proc` filesystem is the standard place to find information about all processes running on a system, and it is the place that the `ps` command gets its information from.

`/proc/<pid>`

Each process has a virtual dirctory in `/proc`, and those directories contain virtual files that hold information about that process. You can use the special shell variable `$$` to find the process ID of your login shell and then look at its information:

```
[root@te-master ~]# echo $$
11421
[root@te-master ~]# ls /proc/11421
attr             cwd       map_files    oom_adj        schedstat  task
autogroup        environ   maps         oom_score      sessionid  timers
auxv             exe       mem          oom_score_adj  setgroups  uid_map
cgroup           fd        mountinfo    pagemap        smaps      wchan
clear_refs       fdinfo    mounts       patch_state    stack
cmdline          gid_map   mountstats   personality    stat
comm             io        net          projid_map     statm
coredump_filter  limits    ns           root           status
cpuset           loginuid  numa_maps    sched          syscall
```

You can look at these individual files to get more information about a process. For example, the `cmdline` file tells you what the process's original command line was

```
[root@te-master ~]# cat /proc/11421/cmdline
-bash
```

Some of these files are special. For example, `exe` appears as a symbolic link to the binary that the process is running:

```
[root@te-master ~]# ls -l /proc/11421/exe
lrwxrwxrwx 1 root root 0 May 31 10:10 /proc/11421/exe -> /usr/bin/bash
```

Some of these files are *really* special. For example, `mem` is a file representing the actual memory being used in the kernel by the process.

Tools like `ps` and `top` use the information in `/proc` to display an overview of the processes on a system, but there is a lot of other information in the directory that can be useful for looking more closely at what a process is doing.

`/proc/cpuinfo`

The `/proc` filesystem contains a variety of other files that provide insight into the kernel's view of a system. One of these is `/proc/cpuinfo`:

```
[root@te-master ~]# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
stepping        : 1
microcode       : 0xb000017
cpu MHz         : 1200.219
cache size      : 20480 KB
physical id     : 0
siblings        : 16
core id         : 0
cpu cores       : 8
apicid          : 0
initial apicid  : 0
fpu             : yes
...
```

This file contains information about every processor on a system, and can be very long on multi-core systems. This is a good place to look to see how many processors your system has, who manufactured them, and what their speeds are.

`/proc/meminfo`

The `/proc/meminfo` file contains more detailed information about system memory usage than `free` provides:

```
[root@te-master ~]# cat /proc/meminfo
MemTotal:       20558264 kB
MemFree:        19296476 kB
MemAvailable:   19500660 kB
Buffers:            2068 kB
```

```
Cached:           191800 kB
SwapCached:            0 kB
Active:           264256 kB
Inactive:         147752 kB
...
```

This file can be useful for getting a more complete view of a system's memory use, especially when you are looking at system preformance and need to know how much total memory is being used and how the kernel is using it.

`/proc/self`

There is a special `/proc` directory called self that points to the `/proc/pid` directory for the current running process.

```
$ ls -l /proc/self
lrwxrwxrwx. 1 root root 0 Jun  4 13:41 /proc/self -> 23371
```

```
$ cat /proc/self/cmdline
cat /proc/self/cmdline
```

That's right, the command is actually the command that inspects the file, because that's the process touching the file. A little confusing, but very handy.

`/proc/self/{mounts,mountinfo}`

In one of the previous sections we looked at our mount list with the `mount` command. This can be dangerous if we have a broken mount. The `mount` comandd could hang forever rather than giving us the information we want. Instead, we can run:

```
$ cat /proc/self/mounts
rootfs / rootfs rw 0 0
sysfs /sys sysfs rw,seclabel,nosuid,nodev,noexec,relatime 0 0
proc /proc proc rw,nosuid,nodev,noexec,relatime 0 0
devtmpfs /dev devtmpfs
rw,seclabel,nosuid,size=10267100k,nr_inodes=2566775,mode=755 0 0
securityfs /sys/kernel/security securityfs rw,nosuid,nodev,noexec,relatime
0 0
tmpfs /dev/shm tmpfs rw,seclabel,nosuid,nodev 0 0
...
```

For even more info, there is:

```
$ cat /proc/self/mountinfo
17 38 0:17 / /sys rw,nosuid,nodev,noexec,relatime shared:6 - sysfs sysfs
rw,seclabel
18 38 0:3 / /proc rw,nosuid,nodev,noexec,relatime shared:5 - proc proc rw
19 38 0:5 / /dev rw,nosuid shared:2 - devtmpfs devtmpfs
rw,seclabel,size=10267100k,nr_inodes=2566775,mode=755
20 17 0:16 / /sys/kernel/security rw,nosuid,nodev,noexec,relatime shared:7
- securityfs securityfs rw
...
```

`/sys`

As we noted above, `/proc` has a virtual directory for each process, but it also has extra virtual files and directories that contain information about the physical system itself. In order to prevent `/proc` from becoming too cluttered while at the same time making space to add more views into the kernel, a new filesystem was added to contain system-related information: `/sys`.

```
[root@te-master ~]# ls -F /sys
block/   class/   devices/   fs/          kernel/  power/
bus/     dev/     firmware/  hypervisor/  module/
```

The `/sys` filesystem is much more hierarchical than /proc and is focused on providing an interface to drivers and hardware devices within the kernel. Interesting directories include:

- `/sys/bus` - Communication busses on the system, including PCI, USB, and memory hardware
- `/sys/fs` - Views of the system as seen by filesystem drivers
- `/sys/devices/system/cpu` - Kernel views of each of the processors on the system

The files in `/sys` are used by various system utilities as an interface to talking to the kernel and can be used to set or read driver parameters after boot time.

To get an idea of what information you can get from `/sys`, try:

```
$ ls -l /sys/class/net
total 0
lrwxrwxrwx. 1 root root 0 Jun  4 13:41 em1 ->
../../devices/pci0000:00/0000:00:04.0/net/em1
lrwxrwxrwx. 1 root root 0 Jun  4 13:41 em2 ->
../../devices/pci0000:00/0000:00:03.0/net/em2
lrwxrwxrwx. 1 root root 0 Jun  4 13:41 ens5 ->
../../devices/pci0000:00/0000:00:05.0/net/ens5
lrwxrwxrwx. 1 root root 0 Jun  4 13:41 lo -> ../../devices/virtual/net/lo
```

This gives a listing of the network devices on the system. You can see that the device names are symbolic links to PCI devices by PCI device ID. Let's take a look inside one:

```
$ ls -l /sys/class/net/em1/
total 0
-r--r--r--. 1 root root 4096 Jun  4 13:41 addr_assign_type
-r--r--r--. 1 root root 4096 Jun  4 13:41 address
-r--r--r--. 1 root root 4096 Jun  5 02:58 addr_len
-r--r--r--. 1 root root 4096 Jun  5 02:58 broadcast
-rw-r--r--. 1 root root 4096 Jun  5 02:58 carrier
-r--r--r--. 1 root root 4096 Jun  5 02:58 carrier_changes
lrwxrwxrwx. 1 root root    0 Jun  4 13:41 device -> ../../../0000:00:04.0
... (lots of files omitted)
```

There is a lot of info about the device here, and some of the special files allow you to control the device by, e.g. writing to them. That's take a look inside `address`:

```
$ cat /sys/class/net/em2/address
de:ad:be:ef:ff:ff
```

This is the MAC address of our ethernet device.

Similar structures exist for many devices and subsystems in the kernel. This can be a great way to inspect deeper system properties, and even write programs or scripts to view and manipulate them.

`/dev`

The "everything is a file" abstraction model extends to directly accessing physical devices, and the files representing devices themselves (as opposed to the kernel's driver files in `/sys`) can be found in `/dev`.

```
[root@te-master ~]# ls -F /dev
autofs           log=                shm/       tty25  tty48  uhid
block/           loop-control        snapshot   tty26  tty49  uinput
bsg/             mapper/             snd/       tty27  tty5   urandom
btrfs-control    mcelog              stderr@    tty28  tty50  usbmon0
bus/             mem                 stdin@     tty29  tty51  usbmon1
char/            mqueue/             stdout@    tty3   tty52  vcs
console          net/                tty        tty30  tty53  vcs1
core@            network_latency     tty0       tty31  tty54  vcs2
cpu/             network_throughput  tty1       tty32  tty55  vcs3
cpu_dma_latency  null                tty10      tty33  tty56  vcs4
crash            nvram               tty11      tty34  tty57  vcs5
disk/            oldmem              tty12      tty35  tty58  vcs6
dri/             port                tty13      tty36  tty59  vcsa
fb0              ppp                 tty14      tty37  tty6   vcsa1
fd@              ptmx                tty15      tty38  tty60  vcsa2
full             ptp0                tty16      tty39  tty61  vcsa3
fuse             pts/                tty17      tty4   tty62  vcsa4
hpet             random              tty18      tty40  tty63  vcsa5
hugepages/       raw/                tty19      tty41  tty7   vcsa6
hwrng            rtc@                tty2       tty42  tty8   vfio/
```

```
infiniband/      rtc0                tty20     tty43  tty9   vga_arbiter
initctl@         sda                 tty21     tty44  ttyS0  vhci
input/           sda1                tty22     tty45  ttyS1  vhost-net
kmsg             sda2                tty23     tty46  ttyS2  zero
knem             sg0                 tty24     tty47  ttyS3
```

The files in `/dev` are connected to their physical devices at a very low level, so you must take care when working with them. For example, `/dev/sda` is the actual boot drive on this system, and copying another file on top of it would most likely make the root filesystem unusable.

A lot of device functions are controlled by making special `ioctl` (input-output control) syscalls on the files in `/dev`. `ioctl` is a file syscall like `write`. It allows an interface like `syscall`, but for specific device functions.

After looking at these three filesystems, it has probably become clear that there is a lot of overlap between the files they contain. This is due to the iterative way in which they were developed - as `/proc` and `/dev` got cluttered with files that didn't have to do with processes or devices, `/sys` was developed to help the situation. But backward compatibility is still important, so legacy files exist in both of the other locations that may not make sense there anymore. Writing perfect software is hard.

Part 4: Logging

Linux systems are dynamic systems that are constantly changing, but most of the tools and files we have looked at so far only look at a single point in time. Systems provide a few logging tools that are useful for finding information about events that have already passed.

dmesg

The `dmesg` command prints out the kernel's log buffer, which is frequently very long. This log starts at boot time and, until it gets too long and is truncated, goes up to the present time.

```
[root@te-master ~]# dmesg
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.10.0-957.12.2.el7.x86_64
(mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5 20150623 (Red Hat
4.8.5-36) (GCC) ) #1 SMP Tue May 14 21:24:32 UTC 2019
[    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-3.10.0-
957.12.2.el7.x86_64 root=UUID=c0c1de66-90c5-4515-8410-7cc166030bd6 ro
[    0.000000] e820: BIOS-provided physical RAM map:
...
[    6.853739] IPv6: ADDRCONF(NETDEV_UP): ib0: link is not ready
[    6.860594] IPv6: ADDRCONF(NETDEV_CHANGE): ib0: link becomes ready
[    7.451210] e1000: em2 NIC Link is Up 1000 Mbps Full Duplex, Flow
Control: RX
[    7.456375] IPv6: ADDRCONF(NETDEV_CHANGE): em2: link becomes ready
[    7.464115] e1000: em1 NIC Link is Up 1000 Mbps Full Duplex, Flow
Control: RX
[    7.477089] e1000: ens5 NIC Link is Up 1000 Mbps Full Duplex, Flow
Control: RX
```

```
[    7.481344] IPv6: ADDRCONF(NETDEV_CHANGE): em1: link becomes ready
[    7.486775] IPv6: ADDRCONF(NETDEV_CHANGE): ens5: link becomes ready
```

The first column of `dmesg` output is a timestamp measured in seconds since the kernel started booting. The rest of each line is free-form text that was provided by the kernel subcomponent that printed it.

The `dmesg` command is a good place to start when you suspect something is wrong with a system at the kernel level. It tends to be fairly quiet after boot time, but the kernel will use this buffer to log any problems or unusual circumstances it sees.

There are a number of ways you can view the `dmesg` output with different options. One particularly handy one is `-H`:

```
[Jun 4 13:41] Initializing cgroup subsys cpuset
[  +0.000000] Initializing cgroup subsys cpu
[  +0.000000] Initializing cgroup subsys cpuacct
[  +0.000000] Linux version 3.10.0-957.12.2.el7.x86_64
(mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5 20150623 (Red Hat
4.8.5-36)
[  +0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-3.10.0-
957.12.2.el7.x86_64 root=UUID=c0c1de66-90c5-4515-8410-7cc166030bd6 ro
[  +0.000000] e820: BIOS-provided physical RAM map:
[  +0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff]
usable
[  +0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff]
reserved
[  +0.000000] BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff]
reserved
```

This gives a more human-readable output, with a pager and color (which you can't see in this guide).

journalctl

While the `dmesg` command prints out the contents of the kernel's log buffer, the `journalctl` command is used to examine the general-purpose logging infrastructure provided by `systemd`. These logs are stored in a special binary format that makes them easy to search through and filter, but makes it impossible to use traditional Unix tools like `grep`, `sed`, and `awk` on them directly.

```
[root@te-master ~]# journalctl
-- Logs begin at Thu 2019-05-30 14:02:19 MDT, end at Sat 2019-06-01
21:31:48 MDT. --
May 30 14:02:19 te-master systemd-journal[162]: Runtime journal is using
8.0M (max allowed 1003.8M, trying to leave 1.4G free of 9.7G available →
current limit 1003.8M).
May 30 14:02:19 te-master kernel: Initializing cgroup subsys cpuset
May 30 14:02:19 te-master kernel: Initializing cgroup subsys cpu
May 30 14:02:19 te-master kernel: Initializing cgroup subsys cpuacct
May 30 14:02:19 te-master kernel: Linux version 3.10.0-957.12.2.el7.x86_64
```

```
(mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5 20150623 (Red Hat
4.8.5-36) (GCC) ) #1 SMP Tue May 14 21:24:32 UTC 2019
...
```

When run with no arguments, `journalctl` will display all of the logs that it knows about. Each line starts with a timestamp, followed by the name of the system the log entry was generated on (`te-master`) and the component that generated it (`systemd-journal` and `kernel` in this example). The rest of the line is then a free-form set of text that was sent by the component as its log message.

Note that, by default, `journalctl` will send its output through a pager (`less`, usually), so you will need to hit the `q` key to exit the display after running this command.

One handy option is to add `-r` to get log entries in reverse order (i.e. most recent on top).

journalctl -f

By adding the `-f` option to `journalctl`, you can *follow* the journal as it is written to. The output will start with a number of recently written lines, at which point the command will pause and wait for more log lines to be written to the journal. It will then print out new lines in real time as they come in.

Since it is impossible to display dynamic text in this guide, you should try this out by following these steps:

1. In one terminal window, run `journalctl -f` on your master node. Let it continue running.
2. In a second terminal window, log in to your master node a second time.
3. You should see log messages generated by your second login appear on your `journalctl -f` output.

journalctl -u

By default, `journalctl` will display the output of all systemd units that wrote to it. You can filter it to only show entries from specific units by using the `-u` option.

```
[root@te-master ~]# journalctl -u sshd
-- Logs begin at Thu 2019-05-30 14:02:19 MDT, end at Sat 2019-06-01
21:39:29 MDT. --
May 30 14:02:25 te-master systemd[1]: Starting OpenSSH server daemon...
May 30 14:02:25 te-master sshd[4311]: Server listening on 0.0.0.0 port 22.
May 30 14:02:25 te-master sshd[4311]: Server listening on :: port 22.
May 30 14:02:25 te-master systemd[1]: Started OpenSSH server daemon.
May 30 14:03:49 te-master sshd[5063]: Accepted publickey for root from
172.16.1.253 port 48810
May 30 14:44:25 te-master sshd[5301]: Accepted publickey for root from
172.16.1.253 port 48812
May 30 14:46:18 te-master sshd[5420]: Accepted publickey for root from
172.16.1.253 port 48814
```

In this example, we are looking only at the output related to the `sshd` unit. Filtering by unit is very useful when you are trying to figure out why a service doesn't appear to be working correctly, or when you want to see what activity has been happening with a service like `ssh`.

journalctl --since and --until

The `journalctl` command can filter in time ranges using the `--since` and `--until` options. These can be used individually, or they can be used together to filter within a very specific window of time.

```
[root@te-master ~]# journalctl --since "2019-05-31 00:00:00" --until
"2019-05-31 12:00:00"
-- Logs begin at Thu 2019-05-30 14:02:19 MDT, end at Sat 2019-06-01
21:46:29 MDT. --
May 31 00:01:01 te-master systemd[1]: Started Session 14 of user root.
May 31 00:01:01 te-master CROND[14801]: (root) CMD (run-parts
/etc/cron.hourly)
May 31 00:01:01 te-master run-parts(/etc/cron.hourly)[14804]: starting
0anacron
May 31 00:01:01 te-master anacron[14810]: Anacron started on 2019-05-31
May 31 00:01:01 te-master anacron[14810]: Normal exit (0 jobs run)
May 31 00:01:01 te-master run-parts(/etc/cron.hourly)[14812]: finished
0anacron
May 31 00:03:27 te-master munged[4917]: Purged 10 credentials from replay
hash
May 31 00:06:27 te-master munged[4917]: Purged 20 credentials from replay
hash
May 31 00:09:27 te-master munged[4917]: Purged 10 credentials from replay
hash
...
```

In this example, we are just looking at log messages that came in between midnight and noon on May 31. If we had left out `--since`, it would have started at the earliest messages available. If we had left out `--until`, it would have ended at the current time.

The `journalctl` command has many, many more ways to filter. You can find more in its man page (`man journalctl`). Note that you can use many commands together to create complex filters. For example, we can use `-u`, `--since`, and `--until` together to find all `ssh` activity between midnight and noon on May 31:

```
[root@te-master ~]# journalctl -u sshd --since "2019-05-31 00:00:00" --
until "2019-05-31 12:00:00"
-- Logs begin at Thu 2019-05-30 14:02:19 MDT, end at Sat 2019-06-01
21:49:29 MDT. --
May 31 09:10:08 te-master sshd[11418]: Accepted publickey for root from
172.16.1.253 port 48818
May 31 09:51:45 te-master sshd[13752]: Accepted publickey for root from
172.16.1.253 port 48820
```

/var/log

While the binary format used by `journald` makes it conventient to filter and search logs, it also makes it difficult to send the logs through other programs for analysis. The `/var/log` directory contains plain text copies of system log files, frequently broken out by the component that generated them:

```
audit/              dmesg               mariadb/            tallylog
boot.log            dmesg.old           messages            tuned/
boot.log-20190531   firewalld           munge/              wtmp
btmp                grubby_prune_debug  ntpstats/           yum.log
conman/             httpd/              secure              yum.log-20190530
conman.log          lastlog             slurm_jobacct.log
cron                maillog             spooler
```

These logs are generated by the `syslog` service, and which logs are written into which files is controlled by its configuration. The overall contents of these files are substantially similar to those found with `journalctl`, so deciding which place to look for information is generally based on how you plan to process the data.

Part 5: System Control

- reboot, halt, systemctl reboot, systemctl poweroff

All good things must come to an end, and you will eventually need to either reboot or shut down one of your systems. These actions can both be done two different ways: using `systemctl`, and using traditional Unix commands. There is no particular advantage to either option - choose what you like the best.

The traditional way

```
[root@te-master ~]# reboot
```

The `reboot` command cleanly restarts your system. If you are logged in over via `ssh`, your connection will immediately drop and you won't be able to log in again until the system has restarted. Make sure your network is configured correctly before doing this so you will be able to log in again!

```
[root@te-master ~]# halt -p
```

The `halt` command shuts down the system, but does not restart it. When the `-p` option is included, it also powers the system off. After running the `halt` command, you will no longer have access to your system until you physically power it back on, either via remote power commands or by pushing the button on its front panel.

The systemd way

```
[root@te-master ~]# systemctl reboot
```

This performs the same action as `reboot`, but uses `systemd` syntax to do it.

```
[root@te-master ~]# systemctl poweroff
```

This performs the same action as `halt -p`, but uses `systemd` syntax to do it.