

System and Administrative Commands

The startup and shutdown scripts in `/etc/rc.d` illustrate the uses of many of these commands. They are usually invoked by `root` and used for system maintenance or file system repairs. The commands must be used with caution, as some of these commands may damage your system.

Users and Groups

users

Show all logged on users. This is the approximate equivalent of `who -q`.

groups

Lists the current user and the groups she belongs to. This corresponds to the `$GROUPS` internal variable, but gives the group names, rather than the numbers.

```
bash$ groups
bozita cdrom cdwriter audio xgrp

bash$ echo $GROUPS
501
```

chown, chgrp

The `chown` command changes the ownership of a file or files. This command is used by `root` to shift file ownership from one user to another. An ordinary user may not change the ownership of files, not even their own files. [\[1\]](#)

```
root# chown bozo *.txt
```

The `chgrp` command changes the *group* ownership of a file or files. You must be owner of the file(s) as well as a member of the destination group (or `root`) to use this operation.

```
chgrp --recursive billybob *.data
# The "billybob" group will now own all the "*.data" files
#+ all the way down the $PWD directory tree (that's what "recursive" means).
```

useradd, userdel

The **useradd** administrative command adds a user account to the system and creates a home directory for that particular user, if so specified. The corresponding **userdel** command removes a user account from the system [\[2\]](#) and deletes associated files.



The **adduser** command is a synonym for **useradd** and is usually a symbolic link to it.

usermod

Modify a user account. Changes may be made to the password, group membership, expiration date, and other attributes of a given user's account. With this command, a user's password may be locked, which has the effect of disabling the account.

groupmod

Modify a given group. The group name and/or ID number may be changed using this command.

id

The **id** command lists the real and effective user IDs and the group IDs of the user associated with the current process. This is the counterpart to the [\\$UID](#), [\\$EUID](#), and [\\$GROUPS](#) internal Bash variables.

```
bash$ id
uid=501(bozo) gid=501(bozo) groups=501(bozo),22(cdrom),80(cdwriter),81(audio)

bash$ echo $UID
501
```



The **id** command shows the *effective* IDs only when they differ from the *real* ones.

Also see [Example 9-5](#).

who

Show all users logged on to the system.

```
bash$ who
bozo tty1 Apr 27 17:45
bozo pts/0 Apr 27 17:46
bozo pts/1 Apr 27 17:47
bozo pts/2 Apr 27 17:49
```

The `-m` gives detailed information about only the current user. Passing any two arguments to **who** is the equivalent of **who -m**, as in **who am i** or **who The Man**.

```
bash$ who -m
localhost.localdomain!bozo pts/2 Apr 27 17:49
```

whoami is similar to **who -m**, but only lists the user name.

```
bash$ whoami
bozo
```

w

Show all logged on users and the processes belonging to them. This is an extended version of **who**. The output of **w** may be piped to **grep** to find a specific user and/or process.

```
bash$ w | grep startx
bozo tty1 - 4:22pm 6:41 4.47s 0.45s startx
```

logname

Show current user's login name (as found in `/var/run/utmp`). This is a near-equivalent to [whoami](#), above.

```
bash$ logname
bozo

bash$ whoami
bozo
```

However...

```
bash$ su
Password: .....

bash# whoami
root
```

```
bash# logname  
bozo
```



While **logname** prints the name of the logged in user, **whoami** gives the name of the user attached to the current process. As we have just seen, sometimes these are not the same.

su

Runs a program or script as a substitute *user*. **su rjones** starts a shell as user *rjones*. A naked **su** defaults to *root*. See [Example A-15](#).

sudo

Runs a command as root (or another user). This may be used in a script, thus permitting a regular user to run the script.

```
#!/bin/bash  
  
# Some commands.  
sudo cp /root/secretfile /home/bozo/secret  
# Some more commands.
```

The file `/etc/sudoers` holds the names of users permitted to invoke **sudo**.

passwd

Sets, changes, or manages a user's password.
The **passwd** command can be used in a script, but *should not* be.

Example 13-1. Setting a new password

```
#!/bin/bash
# setnew-password.sh: For demonstration purposes only.
#           Not a good idea to actually run this script.
# This script must be run as root.

ROOT_UID=0      # Root has $UID 0.
E_WRONG_USER=65 # Not root?

E_NOSUCHUSER=70
SUCCESS=0

if [ "$UID" -ne "$ROOT_UID" ]
then
    echo; echo "Only root can run this script."; echo
    exit $E_WRONG_USER
else
    echo
    echo "You should know better than to run this script, root."
    echo "Even root users get the blues... "
    echo
fi

username=bozo
NEWPASSWORD=security_violation

# Check if bozo lives here.
grep -q "$username" /etc/passwd
if [ $? -ne $SUCCESS ]
then
    echo "User $username does not exist."
    echo "No password changed."
    exit $E_NOSUCHUSER
fi

echo "$NEWPASSWORD" | passwd --stdin "$username"
# The '--stdin' option to 'passwd' permits
#+ getting a new password from stdin (or a pipe).

echo; echo "User $username's password changed!"

# Using the 'passwd' command in a script is dangerous.

exit 0
```

The **passwd** command's `-l`, `-u`, and `-d` options permit locking, unlocking, and deleting a user's password. Only root may use these options.

ac

Show users' logged in time, as read from `/var/log/wtmp`. This is one of the GNU accounting utilities.

```
bash$ ac
      total      68.08
```

last

List *last* logged in users, as read from `/var/log/wtmp`. This command can also show remote logins.

For example, to show the last few times the system rebooted:

```
bash$ last reboot
reboot    system boot  2.6.9-1.667      Fri Feb  4 18:18
(00:02)
  reboot  system boot  2.6.9-1.667      Fri Feb  4 15:20
(01:27)
  reboot  system boot  2.6.9-1.667      Fri Feb  4 12:56
(00:49)
  reboot  system boot  2.6.9-1.667      Thu Feb  3 21:08
(02:17)
  . . .
wtmp begins Tue Feb  1 12:50:09 2005
```

newgrp

Change user's group ID without logging out. This permits access to the new group's files. Since users may be members of multiple groups simultaneously, this command finds little use.

Terminals

tty

Echoes the name of the current user's terminal. Note that each separate *xterm* window counts as a different terminal.

```
bash$ tty
/dev/pts/1
```

stty

Shows and/or changes terminal settings. This complex command, used in a script, can control terminal behavior and the way output displays. See the info page, and study it carefully.

Example 13-2. Setting an erase character

```
#!/bin/bash
# erase.sh: Using "stty" to set an erase character when reading
input.

echo -n "What is your name? "
read name                                # Try to backspace
                                          #+ to erase characters of input.
                                          # Problems?

echo "Your name is $name."

stty erase '#'                            # Set "hashmark" (#) as erase
character.
echo -n "What is your name? "
read name                                # Use # to erase last character
typed.
echo "Your name is $name."

# Warning: Even after the script exits, the new key value
remains set.

exit 0
```

Example 13-3. secret password: Turning off terminal echoing

```
#!/bin/bash
# secret-pw.sh: secret password

echo
echo -n "Enter password "
read passwd
echo "password is $passwd"
echo -n "If someone had been looking over your shoulder, "
echo "your password would have been compromised."

echo && echo # Two line-feeds in an "and list."

stty -echo # Turns off screen echo.

echo -n "Enter password again "
read passwd
echo
echo "password is $passwd"
echo

stty echo # Restores screen echo.

exit 0
```

```
# Do an 'info stty' for more on this useful-but-tricky command.
```

A creative use of **stty** is detecting a user keypress (without hitting **ENTER**).

Example 13-4. Keypress detection

```
#!/bin/bash
# keypress.sh: Detect a user keypress ("hot keys").

echo

old_tty_settings=$(stty -g) # Save old settings (why?).
stty -icanon
Keypress=$(head -c1        # or $(dd bs=1 count=1 2>
/dev/null)                # on non-GNU systems

echo
echo "Key pressed was \"${Keypress}\"."
echo

stty "$old_tty_settings" # Restore old settings.

# Thanks, Stephane Chazelas.

exit 0
```

Also see [Example 9-3](#).

terminals and modes

Normally, a terminal works in the *canonical* mode. When a user hits a key, the resulting character does not immediately go to the program actually running in this terminal. A buffer local to the terminal stores keystrokes. When the user hits the **ENTER** key, this sends all the stored keystrokes to the program running. There is even a basic line editor inside the terminal.

```
bash$ stty -a
speed 9600 baud; rows 36; columns 96; line = 0;
  intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D;
eol = <undef>; eol2 = <undef>;
  start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase =
^W; lnext = ^V; flush = ^O;
  ...
  isig icanon iexten echo echoe echok -echonl -noflsh -
xcase -tostop -echopr
```

Using canonical mode, it is possible to redefine the special keys for the local

terminal line editor.

```
bash$ cat > filexxx
wha<ctl-W>I<ctl-H>foo bar<ctl-U>hello world<ENTER>
<ctl-D>
bash$ cat filexxx
hello world
bash$ wc -c < filexxx
12
```

The process controlling the terminal receives only 12 characters (11 alphabetic ones, plus a newline), although the user hit 26 keys.

In non-canonical ("raw") mode, every key hit (including special editing keys such as **ctl-H**) sends a character immediately to the controlling process.

The Bash prompt disables both `icanon` and `echo`, since it replaces the basic terminal line editor with its own more elaborate one. For example, when you hit **ctl-A** at the Bash prompt, there's no `^A` echoed by the terminal, but Bash gets a `\A` character, interprets it, and moves the cursor to the beginning of the line.

Stéphane Chazelas

setterm

Set certain terminal attributes. This command writes to its terminal's `stdout` a string that changes the behavior of that terminal.

```
bash$ setterm -cursor off
bash$
```

The **setterm** command can be used within a script to change the appearance of text written to `stdout`, although there are certainly [better tools](#) available for this purpose.

```
setterm -bold on
echo bold hello

setterm -bold off
echo normal hello
```

tset

Show or initialize terminal settings. This is a less capable version of **stty**.

```
bash$ tset -r
```

```
Terminal type is xterm-xfree86.  
Kill is control-U (^U).  
Interrupt is control-C (^C).
```

setserial

Set or display serial port parameters. This command must be run by root user and is usually found in a system setup script.


```
# From /etc/pcmcia/serial script:  
  
IRQ=`setserial /dev/$DEVICE | sed -e 's/.*IRQ: //'`  
setserial /dev/$DEVICE irq 0 ; setserial /dev/$DEVICE irq $IRQ
```

getty, agetty

The initialization process for a terminal uses **getty** or **agetty** to set it up for login by a user. These commands are not used within user shell scripts. Their scripting counterpart is **stty**.

mesg


Enables or disables write access to the current user's terminal. Disabling access would prevent another user on the network to [write](#) to the terminal.

 It can be very annoying to have a message about ordering pizza suddenly appear in the middle of the text file you are editing. On a multi-user network, you might therefore wish to disable write access to your terminal when you need to avoid interruptions.

wall

This is an acronym for "[write](#) all", i.e., sending a message to all users at every terminal logged into the network. It is primarily a system administrator's tool, useful, for example, when warning everyone that the system will shortly go down due to a problem (see [Example 17-1](#)).

```
bash$ wall System going down for maintenance in 5 minutes!  
Broadcast message from bozo (pts/1) Sun Jul  8 13:53:27 2001...  
  
System going down for maintenance in 5 minutes!
```

 If write access to a particular terminal has been disabled with **mesg**, then **wall** cannot send a message to it.

Information and Statistics

uname

Output system specifications (OS, kernel version, etc.) to `stdout`. Invoked with the `-a` option, gives verbose system info (see [Example 12-5](#)). The `-s` option shows only the OS type.

```
bash$ uname -a
Linux localhost.localdomain 2.2.15-2.5.0 #1 Sat Feb 5 00:13:43
EST 2000 i686 unknown

bash$ uname -s
Linux
```

arch

Show system architecture. Equivalent to `uname -m`. See [Example 10-26](#).

```
bash$ arch
i686

bash$ uname -m
i686
```

lastcomm

Gives information about previous commands, as stored in the `/var/account/pacct` file. Command name and user name can be specified by options. This is one of the GNU accounting utilities.

lastlog

List the last login time of all system users. This references the `/var/log/lastlog` file.

```
bash$ lastlog
root          tty1          Fri Dec  7 18:43:21 -
0700 2001
  bin          **Never logged in**
  daemon      **Never logged in**
  ...
  bozo        tty1          Sat Dec  8 21:14:29 -
0700 2001

bash$ lastlog | grep root
root          tty1          Fri Dec  7 18:43:21 -
0700 2001
```



This command will fail if the user invoking it does not have read permission for the `/var/log/lastlog` file.

lsuf

List open files. This command outputs a detailed table of all currently open files and gives information about their owner, size, the processes associated with them, and more. Of course, **lsof** may be piped to [grep](#) and/or [awk](#) to parse and analyze its results.

```
bash$ lsof
COMMAND  PID    USER  FD  TYPE   DEVICE  SIZE  NODE
NAME
init      1     root  mem  REG    3,5    30748 30303
/sbin/init
init      1     root  mem  REG    3,5    73120 8069
/lib/ld-2.1.3.so
init      1     root  mem  REG    3,5   931668 8075
/lib/libc-2.1.3.so
cardmgr   213    root  mem  REG    3,5    36956 30357
/sbin/cardmgr
...
```

strace

Diagnostic and debugging tool for tracing system calls and signals. The simplest way of invoking it is **strace COMMAND**.

```
bash$ strace df
execve("/bin/df", ["df"], [/* 45 vars */]) = 0
uname({sys="Linux", node="bozo.localdomain", ...}) = 0
brk(0) = 0x804f5e4
...
```

This is the Linux equivalent of the Solaris **truss** command.

nmap

Network **m**apper and port scanner. This command scans a server to locate open ports and the services associated with those ports. It can also report information about packet filters and firewalls. This is an important security tool for locking down a network against hacking attempts.

```
#!/bin/bash

SERVER=$HOST # localhost.localdomain
(127.0.0.1).
PORT_NUMBER=25 # SMTP port.

nmap $SERVER | grep -w "$PORT_NUMBER" # Is that particular port
open?
# grep -w matches whole words only,
#+ so this wouldn't match port 1025, for example.
```

```
exit 0
# 25/tcp      open      smtp
```

nc

The **nc** (*netcat*) utility is a complete toolkit for connecting to and listening to TCP and UDP ports. It is useful as a diagnostic and testing tool and as a component in simple script-based HTTP clients and servers.

```
bash$ nc localhost.localdomain 25
220 localhost.localdomain ESMTP Sendmail 8.13.1/8.13.1; Thu, 31
Mar 2005 15:41:35 -0700
```

Example 13-5. Checking a remote server for *identd*

```
#!/bin/sh
## Duplicate DaveG's ident-scan thingie using netcat. Oooh,
he'll be p*ssed.
## Args: target port [port port port ...]
## Hose stdout _and_ stderr together.
##
## Advantages: runs slower than ident-scan, giving remote inetd
less cause
##+ for alarm, and only hits the few known daemon ports you
specify.
## Disadvantages: requires numeric-only port args, the output
sleazitude,
##+ and won't work for r-services when coming from high source
ports.
# Script author: Hobbit <hobbit@avian.org>
# Used in ABS Guide with permission.

# -----
E_BADARGS=65      # Need at least two args.
TWO_WINKS=2       # How long to sleep.
THREE_WINKS=3
IDPORT=113        # Authentication "tap ident" port.
RAND1=999
RAND2=31337
TIMEOUT0=9
TIMEOUT1=8
TIMEOUT2=4
# -----

case "${2}" in
    "" ) echo "Need HOST and at least one PORT." ; exit $E_BADARGS
;;
esac

# Ping 'em once and see if they *are* running identd.
nc -z -w $TIMEOUT0 "$1" $IDPORT || { echo "Oops, $1 isn't
running identd." ; exit 0 ; }
# -z scans for listening daemons.
```

```

# -w $TIMEOUT = How long to try to connect.

# Generate a randomish base port.
RP=`expr $$ % $RAND1 + $RAND2`

TRG="$1"
shift

while test "$1" ; do
  nc -v -w $TIMEOUT1 -p ${RP} "$TRG" ${1} < /dev/null >
/dev/null &
  PROC=$!
  sleep $THREE_WINKS
  echo "${1},${RP}" | nc -w $TIMEOUT2 -r "$TRG" $IDPORT 2>&1
  sleep $TWO_WINKS

# Does this look like a lamer script or what . . . ?
# ABS Guide author comments: "It ain't really all that bad,
#+ rather clever, actually."

  kill -HUP $PROC
  RP=`expr ${RP} + 1`
  shift
done

exit $?

# Notes:
# -----

# Try commenting out line 30 and running this script
#+ with "localhost.localdomain 25" as arguments.

# For more of Hobbit's 'nc' example scripts,
#+ look in the documentation:
#+ the /usr/share/doc/nc-X.XX/scripts directory.

```

And, of course, there's Dr. Andrew Tridgell's notorious one-line script in the BitKeeper Affair:

```
echo clone | nc thunk.org 5000 > e2fsprogs.dat
```

free

Shows memory and cache usage in tabular form. The output of this command lends itself to parsing, using [grep](#), [awk](#) or [Perl](#). The **procinfo** command shows all the information that **free** does, and much more.

```

bash$ free

```

	total	used	free	shared
buffers	cached			
Mem:	30504	28624	1880	15820
1608	16376			
-/+ buffers/cache:		10640	19864	

```
Swap:          68540          3128          65412
```

To show unused RAM memory:

```
bash$ free | grep Mem | awk '{ print $4 }'  
1880
```

procinfo

Extract and list information and statistics from the [/proc pseudo-filesystem](#). This gives a very extensive and detailed listing.

```
bash$ procinfo | grep Bootup  
Bootup: Wed Mar 21 15:15:50 2001      Load average: 0.04 0.21 0.34  
3/47 6829
```

lsdev

List devices, that is, show installed hardware.

```
bash$ lsdev  
Device          DMA   IRQ   I/O Ports  
-----  
cascade         4     2  
dma              4     2     0080-008f  
dma1             4     2     0000-001f  
dma2             4     2     00c0-00df  
fpu              4     2     00f0-00ff  
ide0             14    14    01f0-01f7 03f6-03f6  
...
```

du

Show (disk) file usage, recursively. Defaults to current working directory, unless otherwise specified.

```
bash$ du -ach  
1.0k  ./wi.sh  
1.0k  ./tst.sh  
1.0k  ./random.file  
6.0k  .  
6.0k  total
```

df

Shows filesystem usage in tabular form.

```
bash$ df  
Filesystem          1k-blocks      Used Available Use% Mounted  
on  
/dev/hda5           273262         92607   166547   36% /
```

/dev/hda8	222525	123951	87085	59%	/home
/dev/hda7	1408796	1075744	261488	80%	/usr

dmesg

Lists all system bootup messages to `stdout`. Handy for debugging and ascertaining which device drivers were installed and which system interrupts in use. The output of **dmesg** may, of course, be parsed with [grep](#), [sed](#), or [awk](#) from within a script.

```
bash$ dmesg | grep hda
Kernel command line: ro root=/dev/hda2
hda: IBM-DLGA-23080, ATA DISK drive
hda: 6015744 sectors (3080 MB) w/96KiB Cache, CHS=746/128/63
hda: hda1 hda2 hda3 < hda5 hda6 hda7 > hda4
```

stat

Gives detailed and verbose *statistics* on a given file (even a directory or device file) or set of files.

```
bash$ stat test.cru
  File: "test.cru"
  Size: 49970          Allocated Blocks: 100          Filetype:
Regular File
  Mode: (0664/-rw-rw-r--)      Uid: ( 501/ bozo)  Gid: (
501/ bozo)
  Device: 3,8      Inode: 18185      Links: 1
  Access: Sat Jun  2 16:40:24 2001
  Modify: Sat Jun  2 16:40:24 2001
  Change: Sat Jun  2 16:40:24 2001
```

If the target file does not exist, **stat** returns an error message.

```
bash$ stat nonexistent-file
nonexistent-file: No such file or directory
```

vmstat

Display virtual memory statistics.

```
bash$ vmstat
procs          memory      swap          io system
cpu
 r  b  w  swpd   free   buff  cache  si  so   bi   bo   in
cs us  sy id
 0  0  0     0 11040  2636 38952  0  0   33   7  271
88  8  3 89
```


netstat

Show current network statistics and information, such as routing tables and active connections. This utility accesses information in `/proc/net` ([Chapter 27](#)). See [Example 27-3](#).

netstat -r is equivalent to [route](#).

```
bash$ netstat
Active Internet connections (w/o servers)
  Proto Recv-Q Send-Q Local Address           Foreign Address
State
Active UNIX domain sockets (w/o servers)
  Proto RefCnt Flags           Type           State           I-Node Path
unix    11      [ ]             DGRAM          906
/dev/log
unix    3        [ ]             STREAM         CONNECTED      4514
/tmp/.X11-unix/X0
unix    3        [ ]             STREAM         CONNECTED      4513
. . .
```

uptime

Shows how long the system has been running, along with associated statistics.

```
bash$ uptime
10:28pm up 1:57, 3 users, load average: 0.17, 0.34, 0.27
```



A *load average* of 1 or less indicates that the system handles processes immediately. A load average greater than 1 means that processes are being queued. When the load average gets above 3, then system performance is significantly degraded.

hostname

Lists the system's host name. This command sets the host name in an `/etc/rc.d` setup script (`/etc/rc.d/rc.sysinit` or similar). It is equivalent to **uname -n**, and a counterpart to the [\\$HOSTNAME](#) internal variable.

```
bash$ hostname
localhost.localdomain

bash$ echo $HOSTNAME
localhost.localdomain
```

Similar to the **hostname** command are the **domainname**, **dnsdomainname**, **nisdomainname**, and **ypdomainname** commands. Use these to display or set the system DNS or NIS/YP domain name. Various options to **hostname** also perform these functions.

hostid

Echo a 32-bit hexadecimal numerical identifier for the host machine.

```
bash$ hostid  
7f0100
```



This command allegedly fetches a "unique" serial number for a particular system. Certain product registration procedures use this number to brand a particular user license. Unfortunately, **hostid** only returns the machine network address in hexadecimal, with pairs of bytes transposed.

The network address of a typical non-networked Linux machine, is found in `/etc/hosts`.

```
bash$ cat /etc/hosts  
127.0.0.1          localhost.localdomain  
localhost
```

As it happens, transposing the bytes of `127.0.0.1`, we get `0.127.1.0`, which translates in hex to `007f0100`, the exact equivalent of what **hostid** returns, above. There exist only a few million other Linux machines with this identical *hostid*.

sar

Invoking **sar** (System Activity Reporter) gives a very detailed rundown on system statistics. The Santa Cruz Operation ("Old" SCO) released **sar** as Open Source in June, 1999.

This command is not part of the base Linux distribution, but may be obtained as part of the [sysstat utilities](#) package, written by [Sebastien Godard](#).

```
bash$ sar  
Linux 2.4.9 (brooks.seringas.fr)      09/26/03  
  
10:30:00      CPU      %user      %nice      %system      %iowait  
%idle  
10:40:00      all       2.21       10.90      65.48       0.00  
21.41  
10:50:00      all       3.36       0.00      72.36       0.00  
24.28  
11:00:00      all       1.12       0.00      80.77       0.00  
18.11  
Average:      all       2.23       3.63      72.87       0.00  
21.27  
  
14:32:30      LINUX RESTART  
  
15:00:00      CPU      %user      %nice      %system      %iowait  
%idle  
15:10:00      all       8.59       2.40      17.47       0.00  
71.54
```

15:20:00	all	4.07	1.00	11.95	0.00
82.98					
15:30:00	all	0.79	2.94	7.56	0.00
88.71					
Average:	all	6.33	1.70	14.71	0.00
77.26					

readelf

Show information and statistics about a designated *elf* binary. This is part of the *binutils* package.

```
bash$ readelf -h /bin/bash
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
  Class:                   ELF32
  Data:                     2's complement, little
  endian
  Version:                  1 (current)
  OS/ABI:                   UNIX - System V
  ABI Version:              0
  Type:                     EXEC (Executable file)
  . . .
```

size

The **size** [/path/to/binary] command gives the segment sizes of a binary executable or archive file. This is mainly of use to programmers.

```
bash$ size /bin/bash
text    data    bss     dec     hex filename
495971  22496  17392  535859  82d33 /bin/bash
```

System Logs

logger

Appends a user-generated message to the system log (/var/log/messages). You do not have to be root to invoke **logger**.

```
logger Experiencing instability in network connection at 23:10,
05/21.
# Now, do a 'tail /var/log/messages'.
```

By embedding a **logger** command in a script, it is possible to write debugging information to /var/log/messages.

```
logger -t $0 -i Logging at line "$LINENO".
# The "-t" option specifies the tag for the logger entry.
```

```
# The "-i" option records the process ID.

# tail /var/log/message
# ...
# Jul  7 20:48:58 localhost ./test.sh[1712]: Logging at line 3.
```

logrotate

This utility manages the system log files, rotating, compressing, deleting, and/or e-mailing them, as appropriate. This keeps the `/var/log` from getting cluttered with old log files. Usually [cron](#) runs **logrotate** on a daily basis.

Adding an appropriate entry to `/etc/logrotate.conf` makes it possible to manage personal log files, as well as system-wide ones.



Stefano Falsetto has created [rottlog](#), which he considers to be an improved version of **logrotate**.

Job Control

ps

Process statistics: lists currently executing processes by owner and PID (process ID). This is usually invoked with `ax` or `aux` options, and may be piped to [grep](#) or [sed](#) to search for a specific process (see [Example 11-12](#) and [Example 27-2](#)).

```
bash$ ps ax | grep sendmail
295 ?      S        0:00 sendmail: accepting connections on port 25
```

To display system processes in graphical "tree" format: **ps afjx** or **ps ax --forest**.

pgrep, pkill

Combining the **ps** command with [grep](#) or [kill](#).

```
bash$ ps a | grep mingetty
2212 tty2      Ss+      0:00 /sbin/mingetty tty2
 2213 tty3      Ss+      0:00 /sbin/mingetty tty3
 2214 tty4      Ss+      0:00 /sbin/mingetty tty4
 2215 tty5      Ss+      0:00 /sbin/mingetty tty5
 2216 tty6      Ss+      0:00 /sbin/mingetty tty6
4849 pts/2     S+       0:00 grep mingetty

bash$ pgrep mingetty
2212 mingetty
 2213 mingetty
 2214 mingetty
 2215 mingetty
```

```
2216 mingetty
```

pstree

Lists currently executing processes in "tree" format. The `-p` option shows the PIDs, as well as the process names.

top

Continuously updated display of most cpu-intensive processes. The `-b` option displays in text mode, so that the output may be parsed or accessed from a script.

```
bash$ top -b
 8:30pm up 3 min,  3 users,  load average: 0.49, 0.32, 0.13
45 processes: 44 sleeping, 1 running, 0 zombie, 0 stopped
CPU states: 13.6% user,  7.3% system,  0.0% nice, 78.9% idle
Mem:   78396K av,   65468K used,   12928K free,     0K shrd,
2352K buff
Swap:  157208K av,     0K used,  157208K free
37244K cached

   PID USER      PRI  NI  SIZE  RSS SHARE STAT  %CPU  %MEM  TIME
COMMAND
   848 bozo      17   0   996   996   800 R    5.6   1.2   0:00
top
    1 root       8    0   512   512   444 S    0.0   0.6   0:04
init
    2 root       9    0     0     0     0 SW   0.0   0.0   0:00
keventd
...
```

nice

Run a background job with an altered priority. Priorities run from 19 (lowest) to -20 (highest). Only *root* may set the negative (higher) priorities. Related commands are **renice**, **snice**, and **skill**.

nohup

Keeps a command running even after user logs off. The command will run as a foreground process unless followed by `&`. If you use **nohup** within a script, consider coupling it with a [wait](#) to avoid creating an orphan or zombie process.

pidof

Identifies *process ID (PID)* of a running job. Since job control commands, such as [kill](#) and **renice** act on the *PID* of a process (not its name), it is sometimes necessary to identify that *PID*. The **pidof** command is the approximate counterpart to the [\\$PPID](#) internal variable.

```
bash$ pidof xclock
880
```

Example 13-6. pidof helps kill a process

```
#!/bin/bash
# kill-process.sh

NOPROCESS=2

process=xxxyyyzzz # Use nonexistent process.
# For demo purposes only...
# ... don't want to actually kill any actual process with this
script.
#
# If, for example, you wanted to use this script to logoff the
Internet,
#   process=pppd

t=`pidof $process` # Find pid (process id) of $process.
# The pid is needed by 'kill' (can't 'kill' by program name).

if [ -z "$t" ] # If process not present, 'pidof'
returns null.
then
    echo "Process $process was not running."
    echo "Nothing killed."
    exit $NOPROCESS
fi

kill $t # May need 'kill -9' for stubborn
process.

# Need a check here to see if process allowed itself to be
killed.
# Perhaps another " t=`pidof $process` " or ...

# This entire script could be replaced by
#   kill $(pidof -x process_name)
# but it would not be as instructive.

exit 0
```

fuser

Identifies the processes (by PID) that are accessing a given file, set of files, or directory. May also be invoked with the `-k` option, which kills those processes. This has interesting implications for system security, especially in scripts preventing unauthorized users from accessing system services.

```
bash$ fuser -u /usr/bin/vim
/usr/bin/vim: 3207e(bozo)
```

```
bash$ fuser -u /dev/null
/dev/null:          3009(bozo)  3010(bozo)  3197(bozo)
3199(bozo)
```

One important application for **fuser** is when physically inserting or removing storage media, such as CD ROM disks or USB flash drives. Sometimes trying a [umount](#) fails with a device is busy error message. This means that some user(s) and/or process(es) are accessing the device. An **fuser -um /dev/device_name** will clear up the mystery, so you can kill any relevant processes.

```
bash$ umount /mnt/usbdrive
umount: /mnt/usbdrive: device is busy

bash$ fuser -um /dev/usbdrive
/mnt/usbdrive:      1772c(bozo)

bash$ kill -9 1772
bash$ umount /mnt/usbdrive
```

The **fuser** command, invoked with the `-n` option identifies the processes accessing a *port*. This is especially useful in combination with [nmap](#).


```
root# nmap localhost.localdomain
PORT      STATE SERVICE
25/tcp    open  smtp

root# fuser -un tcp 25
25/tcp:          2095(root)

root# ps ax | grep 2095 | grep -v grep
2095 ?          Ss      0:00 sendmail: accepting connections
```

cron

Administrative program scheduler, performing such duties as cleaning up and deleting system log files and updating the slocate database. This is the superuser version of [at](#) (although each user may have their own `crontab` file which can be changed with the **crontab** command). It runs as a [daemon](#) and executes scheduled entries from `/etc/crontab`.

 Some flavors of Linux run **crond**, Matthew Dillon's version of **cron**.

Process Control and Booting

init

The **init** command is the [parent](#) of all processes. Called in the final step of a bootup, **init** determines the runlevel of the system from `/etc/inittab`. Invoked by its alias **telinit**, and by root only.

telinit

Symlinked to **init**, this is a means of changing the system runlevel, usually done for system maintenance or emergency filesystem repairs. Invoked only by root. This command can be dangerous - be certain you understand it well before using!

runlevel

Shows the current and last runlevel, that is, whether the system is halted (runlevel 0), in single-user mode (1), in multi-user mode (2 or 3), in X Windows (5), or rebooting (6). This command accesses the `/var/run/utmp` file.

halt, shutdown, reboot

Command set to shut the system down, usually just prior to a power down.

service

Starts or stops a system *service*. The startup scripts in `/etc/init.d` and `/etc/rc.d` use this command to start services at bootup.

```
root# /sbin/service iptables stop
Flushing firewall rules:          [ OK
]
Setting chains to policy ACCEPT: filter [
OK ]
Unloading iptables modules:      [
OK ]
```

Network

ifconfig

Network *interface configuration* and tuning utility.


```

bash$ ifconfig -a
lo          Link encap:Local Loopback
            inet addr:127.0.0.1  Mask:255.0.0.0
            UP LOOPBACK RUNNING  MTU:16436  Metric:1
            RX packets:10 errors:0 dropped:0 overruns:0 frame:0
            TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:700 (700.0 b)  TX bytes:700 (700.0 b)

```

The **ifconfig** command is most often used at bootup to set up the interfaces, or to shut them down when rebooting.

```

# Code snippets from /etc/rc.d/init.d/network
# ...
# Check that networking is up.
[ ${NETWORKING} = "no" ] && exit 0

[ -x /sbin/ifconfig ] || exit 0
# ...

for i in $interfaces ; do
    if ifconfig $i 2>/dev/null | grep -q "UP" >/dev/null 2>&1 ;
then
    action "Shutting down interface $i: " ./ifdown $i boot
    fi
# The GNU-specific "-q" option to "grep" means "quiet", i.e.,
producing no output.
# Redirecting output to /dev/null is therefore not strictly
necessary.

# ...

echo "Currently active devices:"
echo ` /sbin/ifconfig | grep ^[a-z] | awk '{print $1}' `
#          ^^^^^ should be quoted to prevent
globbing.
# The following also work.
#   echo $(/sbin/ifconfig | awk '/^[a-z]/ { print $1 }')
#   echo $(/sbin/ifconfig | sed -e 's/ .*//')
# Thanks, S.C., for additional comments.

```

See also [Example 29-6](#).

iwconfig

This is the command set for configuring a wireless network. It is the wireless equivalent of **ifconfig**, above.

route

Show info about or make changes to the kernel routing table.

```
bash$ route
Destination      Gateway          Genmask         Flags   MSS
Window  irtt  Iface
pm3-67.bozosisp *                255.255.255.255 UH      40 0
0 ppp0
127.0.0.0        *                255.0.0.0      U       40 0
0 lo
default          pm3-67.bozosisp 0.0.0.0        UG      40 0
0 ppp0
```

chkconfig

Check network configuration. This command lists and manages the network services started at bootup in the `/etc/rc?.d` directory.

Originally a port from IRIX to Red Hat Linux, **chkconfig** may not be part of the core installation of some Linux flavors.

```
bash$ chkconfig --list
atd                0:off  1:off  2:off  3:on   4:on   5:on
6:off
rwhod              0:off  1:off  2:off  3:off  4:off  5:off
6:off
...
```

tcpdump

Network packet "sniffer". This is a tool for analyzing and troubleshooting traffic on a network by dumping packet headers that match specified criteria.

Dump ip packet traffic between hosts *bozoville* and *caduceus*:

```
bash$ tcpdump ip host bozoville and caduceus
```

Of course, the output of **tcpdump** can be parsed, using certain of the previously discussed [text processing utilities](#).

Filesystem

mount

Mount a filesystem, usually on an external device, such as a floppy or CDROM. The file `/etc/fstab` provides a handy listing of available filesystems, partitions, and devices, including options, that may be automatically or manually mounted.

The file `/etc/mntab` shows the currently mounted filesystems and partitions (including the virtual ones, such as `/proc`).

mount -a mounts all filesystems and partitions listed in `/etc/fstab`, except those with a `noauto` option. At bootup, a startup script in `/etc/rc.d` (`rc.sysinit` or something similar) invokes this to get everything mounted.

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
# Mounts CDROM
mount /mnt/cdrom
# Shortcut, if /mnt/cdrom listed in /etc/fstab
```

This versatile command can even mount an ordinary file on a block device, and the file will act as if it were a filesystem. **Mount** accomplishes that by associating the file with a [loopback device](#). One application of this is to mount and examine an ISO9660 image before burning it onto a CDR. [3]

Example 13-7. Checking a CD image

```
# As root...

mkdir /mnt/cdtest # Prepare a mount point, if not already
there.

mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest # Mount
the image.
# "-o loop" option equivalent to "losetup
/dev/loop0"
cd /mnt/cdtest # Now, check the image.
ls -alR # List the files in the directory tree there.
# And so forth.
```

umount

Unmount a currently mounted filesystem. Before physically removing a previously mounted floppy or CDROM disk, the device must be **umounted**, else filesystem corruption may result.

```
umount /mnt/cdrom
# You may now press the eject button and safely remove the disk.
```



The **automount** utility, if properly installed, can mount and unmount floppies or CDROM disks as they are accessed or removed. On laptops with swappable floppy and CDROM drives, this can cause problems, though.

sync

Forces an immediate write of all updated data from buffers to hard drive (synchronize drive with buffers). While not strictly necessary, a **sync** assures the sys admin or user that the data just changed will survive a sudden power failure.

In the olden days, a **sync; sync** (twice, just to make absolutely sure) was a useful precautionary measure before a system reboot.

At times, you may wish to force an immediate buffer flush, as when securely deleting a file (see [Example 12-55](#)) or when the lights begin to flicker.

losetup

Sets up and configures [loopback devices](#).

Example 13-8. Creating a filesystem in a file

```
SIZE=1000000 # 1 meg

head -c $SIZE < /dev/zero > file # Set up file of designated
size.
losetup /dev/loop0 file # Set it up as loopback
device.
mke2fs /dev/loop0 # Create filesystem.
mount -o loop /dev/loop0 /mnt # Mount it.

# Thanks, S.C.
```

mkswap

Creates a swap partition or file. The swap area must subsequently be enabled with **swapon**.

swapon, swapoff

Enable / disable swap partition or file. These commands usually take effect at bootup and shutdown.

mke2fs

Create a Linux ext2 filesystem. This command must be invoked as root.

Example 13-9. Adding a new hard drive

```
#!/bin/bash

# Adding a second hard drive to system.
# Software configuration. Assumes hardware already mounted.
# From an article by the author of this document.
# In issue #38 of "Linux Gazette", http://www.linuxgazette.com.

ROOT_UID=0 # This script must be run as root.
E_NOTROOT=67 # Non-root exit error.

if [ "$UID" -ne "$ROOT_UID" ]
then
```

```

    echo "Must be root to run this script."
    exit $E_NOTROOT
fi

# Use with extreme caution!
# If something goes wrong, you may wipe out your current
# filesystem.

NEWDISK=/dev/hdb          # Assumes /dev/hdb vacant. Check!
MOUNTPOINT=/mnt/newdisk  # Or choose another mount point.

fdisk $NEWDISK
mke2fs -cv $NEWDISK1      # Check for bad blocks & verbose output.
# Note:    /dev/hdb1, *not* /dev/hdb!
mkdir $MOUNTPOINT
chmod 777 $MOUNTPOINT     # Makes new drive accessible to all
# users.

# Now, test...
# mount -t ext2 /dev/hdb1 /mnt/newdisk
# Try creating a directory.
# If it works, umount it, and proceed.

# Final step:
# Add the following line to /etc/fstab.
# /dev/hdb1 /mnt/newdisk ext2 defaults 1 1

exit 0

```

See also [Example 13-8](#) and [Example 28-3](#).

tune2fs

Tune ext2 filesystem. May be used to change filesystem parameters, such as maximum mount count. This must be invoked as root.



This is an extremely dangerous command. Use it at your own risk, as you may inadvertently destroy your filesystem.

dumpe2fs

Dump (list to `stdout`) very verbose filesystem info. This must be invoked as root.

```

root# dumpe2fs /dev/hda7 | grep 'ount count'
dumpe2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
Mount count:                6
Maximum mount count:        20


```

hdparm

List or change hard disk parameters. This command must be invoked as root, and it may be dangerous if misused.

fdisk

Create or change a partition table on a storage device, usually a hard drive. This command must be invoked as root.

 Use this command with extreme caution. If something goes wrong, you may destroy an existing filesystem.


fsck, e2fsck, debugfs

Filesystem check, repair, and debug command set.

fsck: a front end for checking a UNIX filesystem (may invoke other utilities). The actual filesystem type generally defaults to ext2.

e2fsck: ext2 filesystem checker.

debugfs: ext2 filesystem debugger. One of the uses of this versatile, but dangerous command is to (attempt to) recover deleted files. For advanced users only!

 All of these should be invoked as root, and they can damage or destroy a filesystem if misused.

badblocks

Checks for bad blocks (physical media flaws) on a storage device. This command finds use when formatting a newly installed hard drive or testing the integrity of backup media. [\[4\]](#) As an example, **badblocks /dev/fd0** tests a floppy disk.

The **badblocks** command may be invoked destructively (overwrite all data) or in non-destructive read-only mode. If root user owns the device to be tested, as is generally the case, then root must invoke this command.

lsusb, usbmodules

The **lsusb** command lists all USB (Universal Serial Bus) buses and the devices hooked up to them.

The **usbmodules** command outputs information about the driver modules for connected USB devices.

```
root# lsusb
Bus 001 Device 001: ID 0000:0000
Device Descriptor:
```

```

bLength          18
bDescriptorType  1
bcdUSB           1.00
bDeviceClass     9 Hub
bDeviceSubClass  0
bDeviceProtocol  0
bMaxPacketSize0 8
idVendor         0x0000
idProduct        0x0000
. . .

```

mkbootdisk


Creates a boot floppy which can be used to bring up the system if, for example, the MBR (master boot record) becomes corrupted. The **mkbootdisk** command is actually a Bash script, written by Erik Troan, in the `/sbin` directory.

chroot

Change ROOT directory. Normally commands are fetched from `$PATH`, relative to `/`, the default root directory. This changes the root directory to a different one (and also changes the working directory to there). This is useful for security purposes, for instance when the system administrator wishes to restrict certain users, such as those [telnetting](#) in, to a secured portion of the filesystem (this is sometimes referred to as confining a guest user to a "chroot jail"). Note that after a **chroot**, the execution path for system binaries is no longer valid.

A **chroot /opt** would cause references to `/usr/bin` to be translated to `/opt/usr/bin`. Likewise, **chroot /aaa/bbb /bin/ls** would redirect future instances of **ls** to `/aaa/bbb` as the base directory, rather than `/` as is normally the case. An **alias XX 'chroot /aaa/bbb ls'** in a user's `~/.bashrc` effectively restricts which portion of the filesystem she may run command "XX" on.

The **chroot** command is also handy when running from an emergency boot floppy (**chroot** to `/dev/fd0`), or as an option to **lilo** when recovering from a system crash. Other uses include installation from a different filesystem (an [rpm](#) option) or running a readonly filesystem from a CD ROM. Invoke only as root, and use with care.

 It might be necessary to copy certain system files to a *chrooted* directory, since the normal `$PATH` can no longer be relied upon.

lockfile

This utility is part of the **procmail** package (www.procmail.org). It creates a *lock file*, a semaphore file that controls access to a file, device, or resource. The lock file serves as a flag that this particular file, device, or resource is in use by a

particular process ("busy"), and this permits only restricted access (or no access) to other processes.

```
lockfile /home/bozo/lockfiles/$0.lock
# Creates a write-protected lockfile prefixed with the name of
the script.
```

Lock files are used in such applications as protecting system mail folders from simultaneously being changed by multiple users, indicating that a modem port is being accessed, and showing that an instance of Netscape is using its cache. Scripts may check for the existence of a lock file created by a certain process to check if that process is running. Note that if a script attempts to create a lock file that already exists, the script will likely hang.

Normally, applications create and check for lock files in the `/var/lock` directory. [\[5\]](#) A script can test for the presence of a lock file by something like the following.

```
appname=xyzip
# Application "xyzip" created lock file "/var/lock/xyzip.lock".

if [ -e "/var/lock/$appname.lock" ]
then
  ...
```

flock

Much less useful than the **lockfile** command is **flock**. It sets an "advisory" lock on a file and then executes a command while the lock is on. This is to prevent any other process from setting a lock on that file until completion of the specified command.

```
flock $0 cat $0 > lockfile__$0
# Set a lock on the script the above line appears in,
#+ while listing the script to stdout.
```



Unlike **lockfile**, **flock** does *not* automatically create a lock file.

mknod

Creates block or character device files (may be necessary when installing new hardware on the system). The **MAKEDEV** utility has virtually all of the functionality of **mknod**, and is easier to use.

MAKEDEV

Utility for creating device files. It must be run as root, and in the `/dev` directory.

```
root# ./MAKEDEV
```


This is a sort of advanced version of **mknod**.

tmpwatch

Automatically deletes files which have not been accessed within a specified period of time. Usually invoked by [cron](#) to remove stale log files.

Backup

dump, restore

The **dump** command is an elaborate filesystem backup utility, generally used on larger installations and networks. [\[6\]](#) It reads raw disk partitions and writes a backup file in a binary format. Files to be backed up may be saved to a variety of storage media, including disks and tape drives. The **restore** command restores backups made with **dump**.


fdformat

Perform a low-level format on a floppy disk.

System Resources

ulimit

Sets an *upper limit* on use of system resources. Usually invoked with the `-f` option, which sets a limit on file size (**ulimit -f 1000** limits files to 1 meg maximum). The `-t` option limits the coredump size (**ulimit -c 0** eliminates coredumps). Normally, the value of **ulimit** would be set in `/etc/profile` and/or `~/.bash_profile` (see [Appendix G](#)).

 Judicious use of **ulimit** can protect a system against the dreaded *fork bomb*.

```
#!/bin/bash
# This script is for illustrative purposes only.
# Run it at your own peril -- it *will* freeze your
system.

while true # Endless loop.
do
    $0 & # This script invokes itself . . .
        #+ forks an infinite number of times . . .
        #+ until the system freezes up because all
resources exhausted.
done # This is the notorious "sorcerer's
apprentice" scenario.

exit 0 # Will not exit here, because this script
```

```
will never terminate.
```

A **ulimit -Hu XX** (where *XX* is the user process limit) in `/etc/profile` would abort this script when it exceeds the preset limit.

quota

Display user or group disk quotas.

setquota

Set user or group disk quotas from the command line.

umask

User file creation permissions *mask*. Limit the default file attributes for a particular user. All files created by that user take on the attributes specified by **umask**. The (octal) value passed to **umask** defines the file permissions *disabled*. For example, **umask 022** ensures that new files will have at most 755 permissions (777 NAND 022). [7] Of course, the user may later change the attributes of particular files with [chmod](#). The usual practice is to set the value of **umask** in `/etc/profile` and/or `~/.bash_profile` (see [Appendix G](#)).

Example 13-10. Using umask to hide an output file from prying eyes

```
#!/bin/bash
# rot13a.sh: Same as "rot13.sh" script, but writes output to
"secure" file.

# Usage: ./rot13a.sh filename
# or     ./rot13a.sh <filename
# or     ./rot13a.sh and supply keyboard input (stdin)

umask 177                # File creation mask.
                        # Files created by this script
                        #+ will have 600 permissions.

OUTFILE=decrypted.txt    # Results output to file
"decrypted.txt"         #+ which can only be read/written
                        # by invoker of script (or root).

cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M' > $OUTFILE
#   ^^ Input from stdin or a file.   ^^^^^^^^^^^ Output
redirected to file.

exit 0
```

rdev

Get info about or make changes to root device, swap space, or video mode. The functionality of **rdev** has generally been taken over by **lilo**, but **rdev** remains useful for setting up a ram disk. This is a dangerous command, if misused.

Modules

lsmod

List installed kernel modules.

```
bash$ lsmod
Module                Size  Used by
autofs                 9456   2 (autoclean)
opl3                  11376   0
serial_cs             5456   0 (unused)
sb                   34752   0
uart401               6384   0 [sb]
sound                 58368   0 [opl3 sb uart401]
soundlow              464    0 [sound]
soundcore             2800    6 [sb sound]
ds                    6448    2 [serial_cs]
i82365                22928   2
pcmcia_core           45984   0 [serial_cs ds i82365]
```



Doing a **cat /proc/modules** gives the same information.

insmod

Force installation of a kernel module (use **modprobe** instead, when possible). Must be invoked as root.

rmmod

Force unloading of a kernel module. Must be invoked as root.

modprobe

Module loader that is normally invoked automatically in a startup script. Must be invoked as root.

depmod

Creates module dependency file, usually invoked from startup script.

modinfo

Output information about a loadable module.

```
bash$ modinfo hid
filename:      /lib/modules/2.4.20-6/kernel/drivers/usb/hid.o
description:  "USB HID support drivers"
author:       "Andreas Gal, Vojtech Pavlik <vojtech@suse.cz>"
license:      "GPL"
```

Miscellaneous

env

Runs a program or script with certain [environmental variables](#) set or changed (without changing the overall system environment). The `[varname=xxx]` permits changing the environmental variable `varname` for the duration of the script. With no options specified, this command lists all the environmental variable settings.



In Bash and other Bourne shell derivatives, it is possible to set variables in a single command's environment.

```
var1=value1 var2=value2 commandXXX
# $var1 and $var2 set in the environment of
'commandXXX' only.
```



The first line of a script (the "sha-bang" line) may use `env` when the path to the shell or interpreter is unknown.

```
#!/usr/bin/env perl

print "This Perl script will run,\n";
print "even when I don't know where to find Perl.\n";

# Good for portable cross-platform scripts,
# where the Perl binaries may not be in the expected
# place.
# Thanks, S.C.
```

ldd

Show shared lib dependencies for an executable file.

```
bash$ ldd /bin/ls
libc.so.6 => /lib/libc.so.6 (0x4000c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

watch

Run a command repeatedly, at specified time intervals.

The default is two-second intervals, but this may be changed with the `-n` option.

```
watch -n 5 tail /var/log/messages
```

```
# Shows tail end of system log, /var/log/messages, every five seconds.
```

strip

Remove the debugging symbolic references from an executable binary. This decreases its size, but makes debugging it impossible.

This command often occurs in a [Makefile](#), but rarely in a shell script.

nm

List symbols in an unstripped compiled binary.

rdist

Remote distribution client: synchronizes, clones, or backs up a file system on a remote server.

Notes

- [1] This is the case on a Linux machine or a UNIX system with disk quotas.
- [2] The **userdel** command will fail if the particular user being deleted is still logged on.
- [3] For more detail on burning CDRs, see Alex Withers' article, [Creating CDRs](#), in the October, 1999 issue of [Linux Journal](#).
- [4] The `-c` option to [mke2fs](#) also invokes a check for bad blocks.
- [5] Since only *root* has write permission in the `/var/lock` directory, a user script cannot set a lock file there.
- [6] Operators of single-user Linux systems generally prefer something simpler for backups, such as **tar**.
- [7] NAND is the logical *not-and* operator. Its effect is somewhat similar to subtraction.