Contents

Chapter 1

Introduction

1.1 Ways to find installed software

Whatis command is helpful to get brief information about Linux commands or functions. **Whatis** command displays man page single line description for command that matches string passed as a command line argument to **whatis** command. Example:

```
luca@scippane:~$ whatis echo
echo (1)             - display a line of text
```

If we want to search Linux commands or functions information using wild card, then whatis command gives **-w** option. Example:

```
luca@scippane:~$ whatis -w "ab*"
abort (3)            - cause abnormal process termination
abs (3)              - compute the absolute value of an integer
```

If we want to search Linux commands or functions information using regular expressions, then **whatis** command gives **-r** option. Example (all commands ending with ab):

```
luca@scippane:~$ whatis -r "ab$"
anacrontab (5)       - configuration file for anacron
baobab (1)           - A graphical tool to analyze disk usage
crontab (1)          - maintain crontab files for individual users (Vixie Cron)
crontab (5)          - tables for driving cron
dvgrab (1)           - Capture DV or MPEG-2 Transport Stream (HDV) video and audio da
fstab (5)            - static information about the filesystems
inittab (5)          - init daemon configuration
pmxab (1)            - a MusiXTeX preprocessor
scilab (1)           - manual page for Unknown argument: - version
swab (3)             - swap adjacent bytes
tc-stab (8)          - Generic size table manipulations
```

```
XChangeActivePointerGrab (3) – grab the pointer
XcmsCIELab (3)          – Xcms color structure
XtAddGrab (3)           – redirect user input to a modal widget
XtRemoveGrab (3)        – redirect user input to a modal widget
```

which command to find out if a relevant binary is already in your search path.

```
luca@scippane:~$ which gcc
/usr/bin/gcc
```

If which can't find the command you're looking for, try **whereis**; it searches a broader range of system directories and is independent of your shell's search path. Example:

```
luca@scippane:~$ whereis open
open: /bin/open /usr/share/man/man2/open.2.gz /usr/share/man/man1/open.1.gz
```

If we want to locate binary of Linux command, use **"-b"** option.

```
luca@scippane:~$ whereis -b open
open: /bin/open
```

Another alternative is the incredibly useful **locate** command, which consults a precompiled index of the filesystem to locate filenames that match a particular pattern.

1.2 Scripting and the shell

1.2.1 Pipes and Redirection

Every process has at least three communication channels available to it: "standard input" (STDIN), "standard output" (STDOUT), and "standard error" (STDERR).
The kernel sets up these channels on the process's behalf, so the process itself doesn't necessarily know where they lead. They might connect to a terminal win- dow, a file, a network connection, or a channel belonging to another process, to name a few possibilities. Most commands accept their input from STDIN and write their output to STD- OUT. They write error messages to STDERR. This convention lets you string commands together like building blocks to create composite pipelines.
The shell interprets the symbols <, >, and » as instructions to reroute a command's input or output to or from a file. A < symbol connects the command's STDIN to the contents of an existing file. The > and » symbols redirect STDOUT; > replaces the file's existing contents, and » appends to them. For example,the command

```
echo "This is a test message" > /tmp/mymessage
```

stores a single line in the file /tmp/mymessage, creating the file if necessary. To redirect both STDOUT and STDERR to the same place, use the >& symbol. To redirect STDERR only, use 2> .
To connect the STDOUT of one command to the STDIN of another, use the | symbol, commonly known as a pipe. Some examples:

```
ps -ef | grep httpd
```

To execute a second command only if its precursor completes successfully, you can separate the commands with an && symbol. For example,

```
$ lpr /tmp/t2 && rm /tmp/t2
```

removes /tmp/t2 if and only if it is successfully queued for printing.
Conversely, the || symbol executes the following command only if the preceding command fails.
In a script, you can use a backslash to break a command onto multiple lines, helping to distinguish the error-handling code from the rest of the command pipeline:

```
cp --preserve --recursive /etc/* /spare/backup \
|| echo "Did NOT make backup"
```

For the converse effect,multiple commands combined onto one line, you can use a semicolon as a statement separator.

1.2.2 Variables and Quoting

Variable names are unmarked in assignments but prefixed with a dollar sign when their values are referenced. For example:

```
luca@scippane:~$ etcdir='/etc'
luca@scippane:~$ echo $etchdir
/etc
```

Do not put spaces around the = symbol or the shell will mistake your variable name for a command name.
When referencing a variable, you can surround its name with curly braces to clarify to the parser and to human readers where the variable name stops and other text begins; for example, ${etcdir} instead of just $etcdir.
Variable names are case sensitive.
Environment variables are automatically imported into bash's variable namespace, so they can be set and read with the standard syntax. Use **export** *varname* to promote a shell variable to an environment variable. Commands for environment variables that you want to set up at login time should be included in your /.profile or /.bash_ profile file.
The shell treats strings enclosed in single and double quotes similarly, except that double-quoted strings are subject to globbing (the expansion of filename matching metacharacters such as * and ?) and variable expansion. For example:

```
luca@scippane:~$ myland="Pennsylvania Duch"
luca@scippane:~$ echo "I speak $myland"
I speak Pennsylvania Duch
luca@scippane:~$ echo 'I speak $myland'
I speak $myland
```

Back quotes, also known as back-ticks, are treated similarly to double quotes, but they have the additional effect of executing the contents of the string as a shell command and replacing the string with the command's output. For example,

```
luca@scippane:~$ echo "There are `wc -l /etc/passwd` lines in the password
There are 88 /etc/passwd lines in the password file
```

1.2.3 Common Filter Commands

Any well-behaved command that reads STDIN and writes STDOUT can be used as a filter (that is, a component of a pipeline) to process data.

cut: *separate lines into fields*
The cut command prints selected portions of its input lines.

```
cut -d: -f7 < /etc/passwd | sort -u
```

The cut command picks out the path to each user's shell from /etc/passwd. The list of shells is then sent through sort -u to produce a sorted list of unique values.
sort: *sort lines*
sort sorts its input lines.

Opt	Meaning
-b	Ignore leading whitespace
-f	Case insensitive sorting
-k	Specify the columns which form the sort key
-n	Compare fields as integer numbers
-r	Reverse sort order
-t	Set field separator (The default is white space)
-u	Output unique records only

Examples:

```
sort -t: -k3,3 -n /etc/group
root:x:0:
bin:x:1:daemon
daemon:x:2:

sort -t: -k3,3 /etc/group
root:x:0:
```

```
bin:x:1:daemon
users:x:100:
```

uniq: *print unique lines*
uniq is similar in spirit to sort -u, but it has some useful options that sort does not emulate: **-c** to count the number of instances of each line, **-d** to show only duplicated lines, and **-u** to show only nonduplicated lines. The input must be presorted, usually by being run through sort.
For example the command below shows that 50 users have */bin/bash* as their login shell and that 18 have */bin/false* and so on.

```
cut -d: -f7 /etc/passwd | sort | uniq -c
     50 /bin/bash
     18 /bin/false
      2 /bin/sh
      1 /bin/sync
     17 /usr/sbin/nologin
```

wc: *count lines, words, and characters*
Counting the number of lines, words, and characters in a file is another common operation, and the **wc** (word count) command is a convenient way of doing this. Run without options, it displays all three counts:

```
luca@scippane:~$ wc /etc/passwd
  88  114 5032 /etc/passwd
```

In the context of scripting, it is more common to supply a **-l**, **-w**, or **-c** option to make wc's output consist of a single number.
tee: *copy input to two places* A command pipeline is typically linear, but it's often helpful to tap into the data stream and send a copy to a file or to the terminal window. You can do this with the **tee** command, which sends its standard input both to standard out and to a file that you specify on the command line. For example:

```
find /lib/modules -name core | tee /dev/tty | wc -l
```

prints both the pathnames of files named core and a count of the number of core files that were found.

head and **tail**: *read the beginning or end of a file*
These commands display ten lines by default, but you can include a command-line option to specify how many lines you want to see.
tail also has a nifty **-f** option that's particularly useful for sysadmins. Instead of exiting immediately after printing the requested number of lines, **tail -f** waits for new lines to be added to the end of the file and prints them as they appear; great for monitoring log files.

grep: *search text*
grep searches its input text and prints the lines that match a given pattern. Its

name is based on the g/regular-expression/p command from the old **ed** editor that came with the earliest versions of UNIX (and still does).
Like most filters, grep has many options, including **-c** to print a count of matching lines, **-i** to ignore case when matching, and **-v** to print nonmatching (rather than matching) lines. Another useful option is **-l** (lowercase L), which makes **grep** print only the names of matching files rather than printing each line that matches.
For example:

```
$ sudo grep -l mdadm /var/log/*
/var/log/auth.log
/var/log/syslog.0
```

shows that log entries from mdadm have appeared in two different log files.

1.2.4 Regular Expressions

Regular expressions are supported by most modern languages. They are also used by UNIX commands such as **grep** and **vi**.
The filename matching and expansion performed by the shell when it interprets command lines such as **wc -l *.pl** is not a form of regex matching. It's a different system called "shell globbing" and it uses a different and simpler syntax.
Regular expressions reached the apex of their power and perfection in **Perl**.

The Matching Process Code that evaluates a regular expression attempts to match a single given text **string** to a single given **pattern**.
For the matcher to declare success, the entire search pattern must match a contiguous section of the search text. However, the pattern can match at any position.

Literal Characters In general, characters in a regular expression match themselves.

Special Characters The table below shows the meanings of some common special symbols that can appear in regular expressions. These are just the basics; there are many, many more.

Symbol	**What it matches or does**
.	Matches any character
[*chars*]	Matches any character from a given set
[^*chars*]	Matches any character NOT in a given set
^	Matches the beginning of a line
&	Matches the beginning of a line
\w	Matches any word character
\s	Matches any whitespace character
\d	Matches any digit
|	Matches either the element to its left or the one to its right
(expr)	Limits scope, groups elements, allows matches to be captured
?	Allows zero or one match of the preceding element
*	Allows zero, one, or many matches of the preceding element
+	Allows one or more matches of the preceding element
{n}	Matches exactly n instances of the preceding element
{min,}	Matches at least min instances (note the comma)
{min,max}	Matches any number of instances from min to max

Many special constructs, such as + and | , affect the matching of the "thing" to their left or right. In general, a "thing" is a single character, a subpattern enclosed in parentheses, or a character class enclosed in square brackets.
If you want to limit the scope of the vertical bar, enclose the bar and both things in their own set of parentheses. For example,

```
I am the (walrus|egg man)\.
```

matches either "I am the walrus." or "I am the egg man.". This example also demonstrates escaping of special characters (here, the dot).

1.3 Controlling Processes

A process consists of an address space and a set of data structures within the kernel. The address space is a set of memory pages that the kernel has marked for the process's use.

1.3.1 PID - Process ID Number

The kernel assigns a unique ID number to every process.Most commands and system calls that manipulate processes require you to specify a PID to identify the target of the operation. PIDs are assigned in order as processes are created.

1.3.2 PPID: Process Parent ID

When a process is cloned, the original process is referred to as the parent, and the copy is called the child. The PPID attribute of a process is the PID of the parent from which it was cloned.

1.3.3 UID and EUID: Real and Effective user ID

A process's UID is the user identification number of the person who created it, or more accurately, it is a copy of the UID value of the parent process. Usually, only the creator (aka the "owner") and the superuser can manipulate a process. The EUID is the "effective" user ID, an extra UID used to determine what resources and files a process has permission to access at any given moment.

1.3.4 GID and EGID: Real and Effective Group ID

The GID is the group identification number of a process. The EGID is related to the GID in the same way that the EUID is related to the UID in that it can be "upgraded" by the execution of a setgid program.

1.3.5 Signals

Signals are process-level interrupt requests.
When a signal is received, one of two things can happen. If the receiving process has designated a handler routine for that particular signal, the handler is called with information about the context in which the signal was delivered. Otherwise, the kernel takes some default action on behalf of the process. Some signals with which all administrators should be familiar are:

No	Name	Description	Default
1	HUP	Hangup	Terminate
2	INT	Interrupt	Terminate
3	QUIT	Quit	Terminate
9	KILL	Kill	Terminate
11	SEGV	Segmentation Fault	Terminate
15	TERM	Software Termination	Terminate

1.3.6 KILL: Send Signals

As its name implies, the **kill** command is most often used to terminate a process. **kill** can send any signal, but by default it sends a TERM. kill can be used by normal users on their own processes or by root on any process. The syntax is:

```
kill [-signal] pid
```

where *signal* is the number or symbolic name of the signal to be sent (as shown in Table above) and *pid* is the process identification number of the target process. The command

```
kill -9 pid
```

"guarantees" that the process will die because signal 9, KILL, cannot be caught.

Under Linux, **killall** kills processes by name.

1.3.7 Process States

You need to be aware of the four execution states listed in the table below:

State	Meaning
Runnable	The process can be executed
Sleeping	The process is waiting for some resources
Zombie	The process is trying to die
Stopped	The process is suspended (not allowed to execute)

A runnable process is ready to execute whenever CPU time is available.
It has acquired all the resources it needs and is just waiting for CPU time to process its data.
Sleeping processes are waiting for a specific event to occur.
Zombies are processes that have finished execution but have not yet had their status collected.
Stopped processes are administratively forbidden to run.

1.3.8 PS: Monitor Processes

ps is the system administrator's main tool for monitoring processes.
ps can show the PID, UID, priority, and control terminal of processes. It also gives information about how much memory a process is using, how much CPU time it has consumed, and its current status.
On Linux and AIX, you can obtain a useful overview of all the processes running on the system with **ps aux**. The **a** option means to show all processes, **x** means to show even processes that don't have a control terminal, and **u** selects the "user oriented" output format.
Another useful set of arguments for Linux and AIX is **lax**, which provides more technical information (**l** selects the "long" output format).

1.3.9 Dynamic Monitoring

top is a free utility that runs on many systems and provides a regularly updated summary of active processes and their use of resources.
By default, the display updates every 10 seconds. The most CPU-consumptive processes appear at the top.

1.3.10 The /Proc Filesystem

The Linux versions of **ps** and **top** read their process status information from the /**proc** directory, a pseudo-filesystem in which the kernel exposes a variety of interesting information about the system's state.

1.3.11 Trace Signals and System Calls

Linux lets you directly observe a process with the **strace** command, which shows every system call the process makes and every signal it receives.