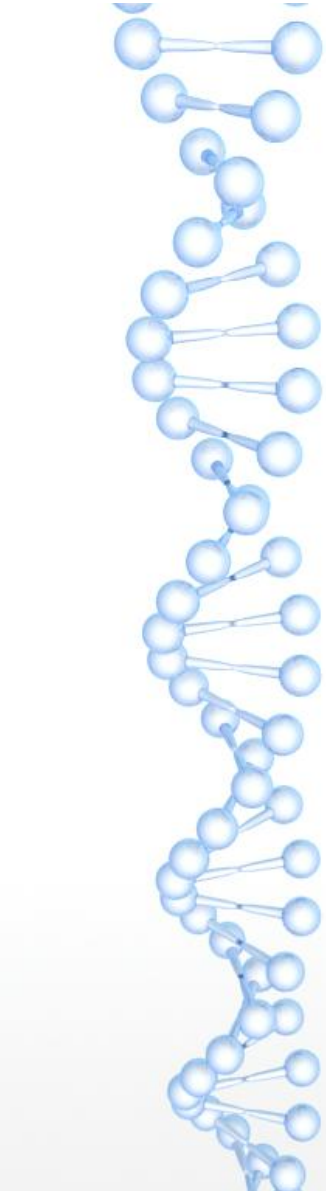


UNIVERSITY OF TORONTO

Advanced Linux command-line interface

SciNet

Shells

Exploring Your Linux Shell Options

Linux provides a range of options for shells.

bash

The GNU Bourne Again Shell (bash) is based on the earlier Bourne shell for Unix but extends it in several ways. In Linux, bash is the most common default shell for user accounts, and it's the one emphasized in this course.

tcsh

This shell is based on the earlier C shell (csh). It's a fairly popular shell in some circles, but no major Linux distributions make it the default shell.

csh

The original C shell isn't much used on Linux, but if a user is familiar with csh, tcsh makes a good substitute.

ksh

The Korn shell (ksh) was designed to take the best features of the Bourne shell and the C shell and extend them further.

zsh

The Z shell (zsh) takes shell evolution further than the Korn Shell, incorporating features from earlier shells and adding still more. zsh is the new default shell on macOS.

This course is about bash. This is the default shell in Linux and is the one we will use in this course. For now on the terms "shell" and "bash" refer to the same.

Shells

Exploring Your Linux Shell Options

PATH

PATH is a shell variable, also called an environment variable. It holds the search path for commands. It is a colon-separated list of directories in which the shell looks for commands. A common value is:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin
```

PATH is slightly different for the root user:

```
# echo $PATH
/sbin:/usr/sbin:/usr/local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/games:/root/bin
```

Shells

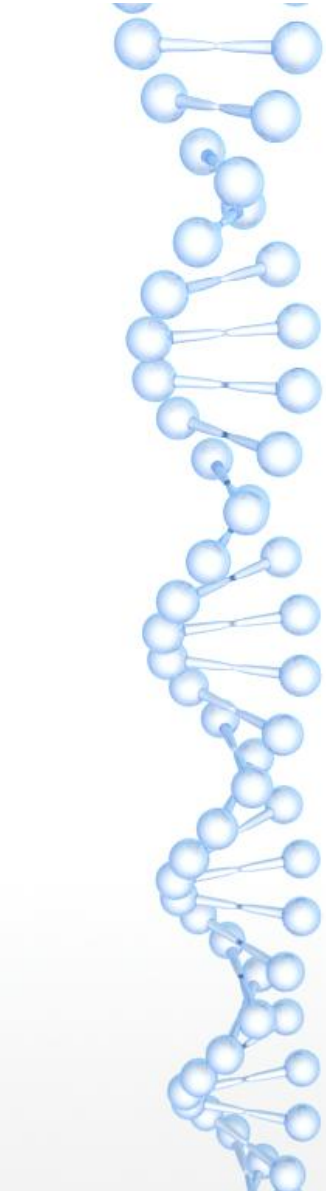
Exploring Your Linux Shell Options

PATH

When you type a command in the command-line, the shell looks for a file in the directories listed in the PATH variable, in the same order as they are in the variable. When the shell finds the first, it runs it.

If you want to know where the executable file is located, you can run a command called *which*:

```
$ which grep
/bin/grep
$ which nocommand
which: no nocommand in (/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin)
```



Exploring Linux command-line tools

Performing Some Shell Command Tricks

Command completion

Many users find typing commands to be tedious and error prone. This is particularly true of slow or sloppy typists. For this reason, Linux bash shell include various tools that can help speed up operations.

The first of these is ***command completion****:* type part of a command or (as an option to a command) a filename, and then press the Tab key. The shell tries to fill in the rest of the command or the filename. If just one command or filename matches the characters you've typed so far, the shell fills it in and places a space after it. If the characters you've typed don't uniquely identify a command or filename, the shell fills in what it can and then stops. Depending on the shell and its configuration, it may beep. If you press the Tab key again, the system responds by displaying the possible completions. You can then type another character or two and, if you haven't completed the command or filename, press the Tab key again to have the process repeat.

Exploring Linux command-line tools

Performing Some Shell Command Tricks

history

This is, by far, the most powerful tool of the command-line. The history keeps a record of every command you type (stored in ~/.bash_history). If you've typed a long command recently and want to use it again, or use a minor variant of it, you can pull the command out of the history.

The simplest way to do this is to press the Up arrow key on your keyboard; this brings up the previous command. Pressing the Up arrow key repeatedly moves through multiple commands so you can find the one you want. If you overshoot, press the Down arrow key to move down the history.

Exploring Linux command-line tools

Performing Some Shell Command Tricks

history (continued)

Another way to use the command history is to search through it. Press Ctrl+R to begin a backward (reverse) search, which is what you probably want, and begin typing characters that should be unique to the command you want to find.

The characters you type need not be the ones that begin the command; they can exist anywhere in the command. You can either keep typing until you find the correct command or, after you've typed a few characters, press Ctrl+R repeatedly until you find the one you want.

Exploring Linux command-line tools

Performing Some Shell Command Tricks

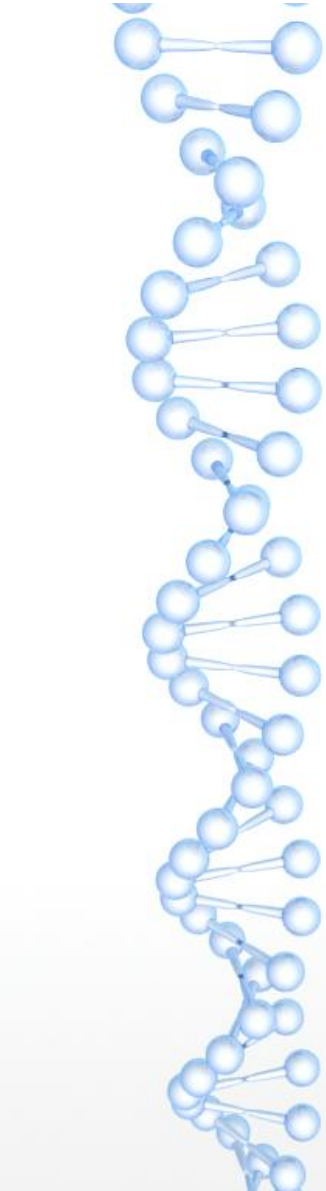
history (continued)

Frequently, after finding a command in the history, you want to edit it. The bash shell, provides editing features modelled after those of the Emacs editor:

Editing the command line

Move within the line

Press Ctrl+A or Ctrl+E to move the cursor to the start or end of the line, respectively. The Left and Right arrow keys move within the line a character at a time. Pressing Ctrl plus the Left or Right arrow key (Alt-left arrow or Alt-right arrow in the Mac) moves backward or forward a word at a time, as does pressing Esc and then B or F.



Exploring Linux command-line tools

Performing Some Shell Command Tricks

history (continued)

Editing the command line (continued)

Delete text Pressing

Ctrl+D or the Delete key deletes the character under the cursor, whereas pressing the Backspace key deletes the character to the left of the cursor. Pressing Ctrl+K deletes all text from the cursor to the end of the line. Pressing Ctrl+X and then Backspace deletes all the text from the cursor to the beginning of the line. Pressing Ctrl-U deletes all text from the cursor to the beginning of the line.

Transpose text

Pressing Ctrl+T transposes the character before the cursor with the character under the cursor. Pressing Esc and then T transposes the two words immediately before (or under) the cursor.

Exploring Linux command-line tools

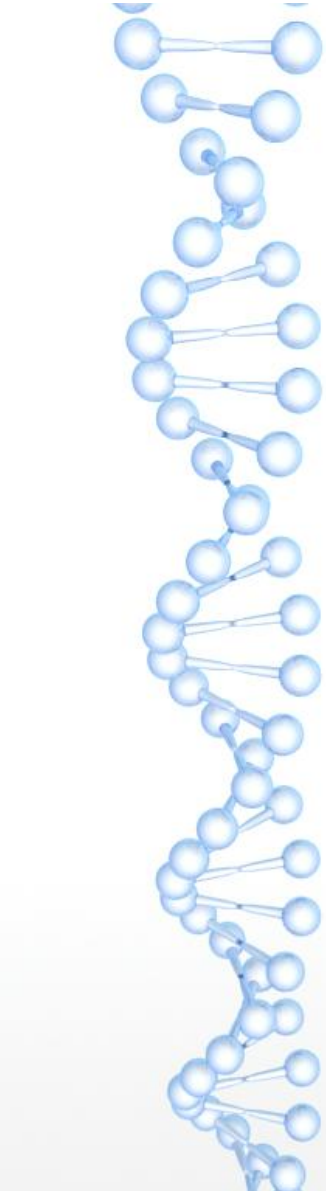
Performing Some Shell Command Tricks

history (continued)

Editing the command line (continued)

Change case

Pressing Esc and then U converts text from the cursor to the end of the word to uppercase. Pressing Esc and then L converts text from the cursor to the end of the word to lowercase. Pressing Esc and then C converts the letter under the cursor (or the first letter of the next word) to uppercase, leaving the rest of the word unaffected.



Exploring Linux command-line tools

Performing Some Shell Command Tricks

history (continued)

These editing commands are just the most useful ones supported by bash history; consult its ***man page*** to learn about many more obscure editing features. In practice, you're likely to make heavy use of command and filename completion, the command history, and perhaps a few editing features.

The *history* command provides an interface to view and manage the history.

Typing *history* alone displays all the commands in the history (typically the latest 500 commands); adding a number causes only that number of the latest commands to appear. Typing *history -c* clears the history, which can be handy if you've recently typed commands you'd rather not have discovered by others (such as commands that include passwords).

Exploring Linux command-line tools

Performing Some Shell Command Tricks

Exercise

Editing Commands

To experiment with your shell's completion and command-line editing tools, follow these steps:

1. Log in as an ordinary user.
2. Create a temporary directory by typing *mkdir test.*
3. Change into the test directory by typing *cd test*.
4. Create a few temporary files by typing *touch one two three*. This command creates three empty files named one, two, and three.
5. Type *ls -l t*, and without pressing the Enter key, press the Tab key. The system may beep at you or display *two three*. If it doesn't display two three, press the Tab key again, and it should do so. This reveals that either two or three is a valid completion to your command, because these are the two files in the test directory whose filenames begin with the letter t.

Exploring Linux command-line tools

Performing Some Shell Command Tricks

Exercise

Editing Commands (continued)

7. Type h, and again without pressing the Enter key, press the Tab key. The system should complete the command (ls -l three), at which point you can press the Enter key to execute it. (You'll see information on the file.)
8. Press the Up arrow key. You should see the ls -l three command appear on the command line.
9. Press Ctrl+A to move the cursor to the beginning of the line.
10. Press the Right arrow key once, and type es (without pressing the Enter key). The command line should now read *less -l three*.
11. Press the Right arrow key once, and press the Delete key three times. The command should now read less three. Press the Enter key to execute the command. (Note that you can do so even though the cursor isn't at the end of the line.) This invokes the less pager on the three file. (The less pager is described more fully later, in "Getting Help.") Because this file is empty, you'll see a mostly empty screen.
12. Press the Q key to exit from the less pager.

Exploring Linux command-line tools

UNIVERSITY OF TORONTO

Shortcuts

aliases

A bash alias is essentially nothing more than a keyboard shortcut, an abbreviation, a means of avoiding typing a long command sequence. If, for example, we include:

```
alias lm="ls -l | more"
```

in the ~/.bashrc file, then each ***lm*** typed at the command-line will automatically be replaced by a **ls -l | more**. This can save a great deal of typing at the command-line and avoid having to remember complex combinations of commands and options. Setting alias **rm="rm -i"** (interactive mode delete) may save a good deal of grief, since it can prevent inadvertently deleting important files.

SciNet

Exploring Linux command-line tools

Getting Help

man

Linux provides a text-based help system known as **man**. This command's name is short for manual. For instance, to learn about man itself, you can type *man man*. The result is a description of the man command.

The man utility uses the *less* pager to display information. We will talk more about *less* later.

Exploring Linux command-line tools

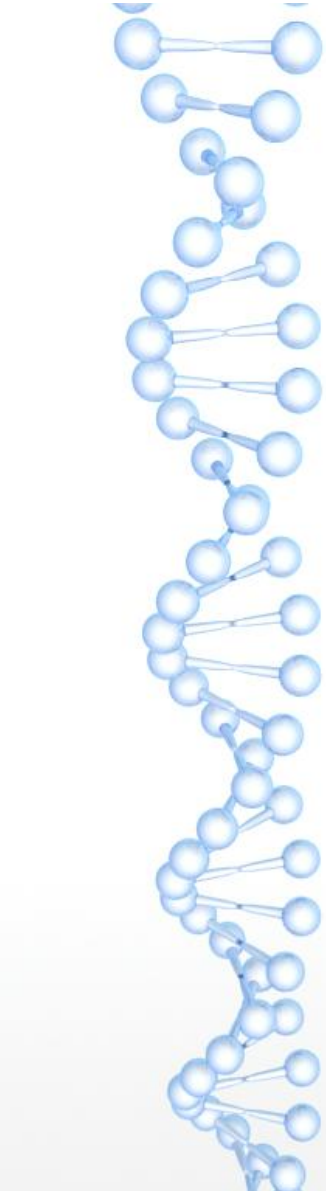
Getting Help

apropos

apropos - search the manual page names and descriptions.

Each manual page has a short description available within it. apropos searches the descriptions for instances of a keyword.

```
$ apropos grep

bzgrep (1)           - search possibly bzip2 compressed files for a regular expression
egrep (1)            - print lines matching a pattern
fgrep (1)            - print lines matching a pattern
grep (1)             - print lines matching a pattern
grep (1p)            - search a file for a pattern
grep-changelog (1)   - print ChangeLog entries matching criteria
msggrep (1)          - pattern matching on message catalog
pgrep (1)            - look up or signal processes based on name and other attributes
pm-utils-bugreport-info.sh (8) - Print pm-utils bug report
xzegrep (1)          - search compressed files for a regular expression
xzfgrep (1)          - search compressed files for a regular expression
xzgrep (1)           - search compressed files for a regular expression
zgrep (1)            - search possibly compressed files for a regular expression
zipgrep (1)          - search files in a ZIP archive for lines matching a pattern
```



UNIVERSITY OF TORONTO

Exploring Linux command-line tools

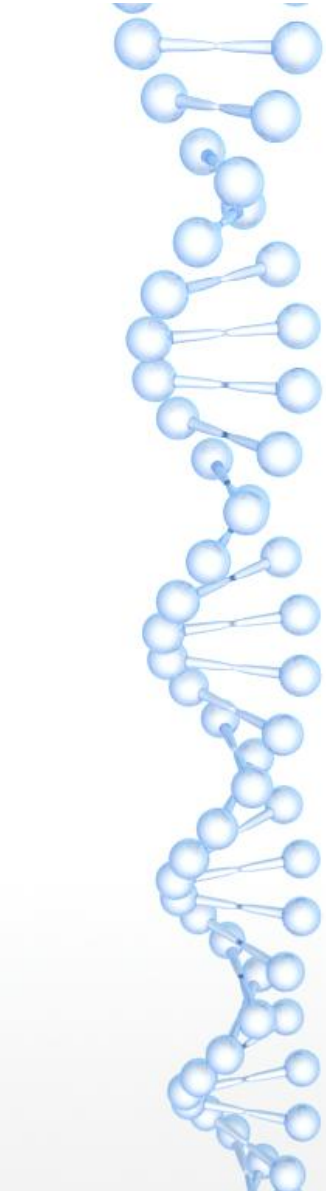
Using Streams, Redirection, and Pipes

Streams, redirection, and pipes

Streams, redirection, and pipes are very powerful command-line tools in Linux. Linux treats the input to and output from programs as a stream, which is a data entity that can be manipulated.

Ordinarily, input comes from the keyboard and output goes to the screen. You can redirect these input and output streams to come from or go to other sources, though, such as files. Similarly, you can **pipe** the output of one program into another program. These facilities can be great tools to tie together multiple programs or commands.

SciNet



Exploring Linux command-line tools

Using Streams, Redirection, and Pipes

Exploring Types of Streams

To begin understanding redirection and pipes, you must first understand the different types of input and output streams. Three are most important for this topic:

Standard input

Programs accept keyboard input via standard input, or stdin. In most cases, this is the data that comes into the computer from a keyboard.

Standard output

Text-mode programs send most data to their users via standard output (a.k.a. stdout), which is normally displayed on the screen, either in a full-screen text-mode session or in a GUI window such as an xterm.

Standard error

Linux provides a second type of output stream, known as standard error, or stderr. This output stream is intended to carry high-priority information such as error messages. Ordinarily, standard error is sent to the same output device as standard output, so you can't easily tell them apart. You can redirect one independently of the other, though, which can be handy. For instance, you can redirect standard error to a file while leaving standard output going to the screen so that you can interact with the program and then study the error messages later.

Exploring Linux command-line tools

Redirection

Redirecting Input and Output

To redirect input or output, you use symbols following the command, including any options it takes. For instance, to redirect the output of the *echo* command, you would type something like this:

```
$ echo $HISTCONTROL > histcontrol.txt
```

The result is that the file histcontrol*.txt* contains the output of the command. Redirection operators exist to achieve several effects, as summarized in the next slide:

Exploring Linux command-line tools

Redirection

Common Redirection Operators

Redirection Operator	Effect
>	Creates a new file containing standard output. If the specified file exists, it's overwritten.
>>	Appends standard output to the existing file. If the specified file doesn't exist, it's created.
2>	Creates a new file containing standard error. If the specified file exists, it's overwritten.
2>>	Appends standard error to the existing file. If the specified file doesn't exist, it's created.
&>	Creates a new file containing both standard output and standard error. If the specified file exists, it's overwritten.
<	Sends the contents of the specified file to be used as standard input
<<	Accepts text on the following lines as standard input.
<>	Causes the specified file to be used for both standard input and standard output.

Exploring Linux command-line tools

Pipes

Piping Data Between Programs

Programs can frequently operate on other programs' outputs.

For instance, you might use a text-filtering command, such as *grep,* to manipulate text output by another program.

The solution is to use data pipes. A pipe redirects the first program's standard output to the second program's standard input and is denoted by a vertical bar (|):

```
$ first | second
```

Exploring Linux command-line tools

Pipes

Piping Data Between Programs (continued)

For instance, suppose that *first* generates some system statistics, such as system uptime, CPU use, number of users logged in, and so on. This output might be lengthy, so you want to trim it a bit.

You might therefore use second, which could be a script or command that echoes from its standard input only the information in which you're interested. The *grep* command is often used in this role.

Pipes can be used in sequences of arbitrary length:

```
$ first | second | third | fourth | fifth | sixth [...]
```

Exploring Linux command-line tools

Processing Text Using Filters

Many simple commands are available to manipulate text. These commands accomplish tasks of various types, such as combining files, transforming the data in files, formatting text, displaying text, and summarizing data.

File-Combining Commands

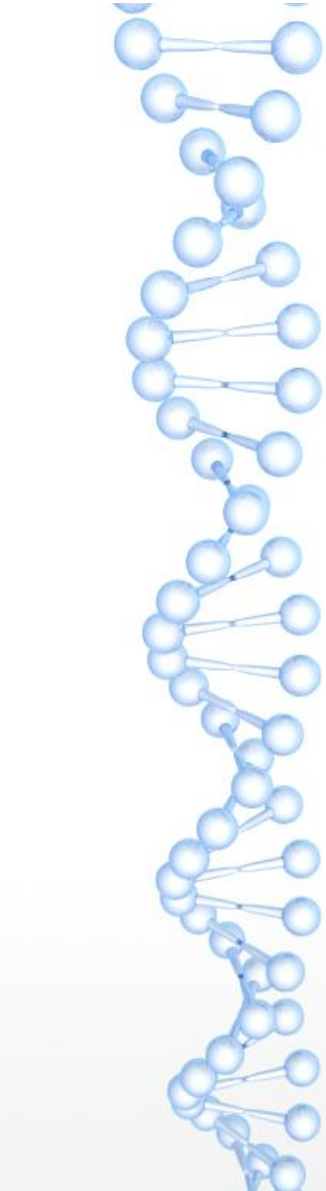
Combining Files with *cat*

The cat command's name is short for *concatenate*, and this tool does just that: It links together an arbitrary number of files end to end and sends the result to standard output.

By combining cat with output redirection, you can quickly combine two files into one:

```
$ cat first.txt second.txt > combined.txt
```

Although cat is officially a tool for combining files, it's also commonly used to display the contents of a short file. If you type only one filename as an option, cat displays that file. This is a great way to review short files; but for long files, you're better off using a full-fledged pager command, such as *more* or *less*.



Exploring Linux command-line tools

Processing Text Using Filters

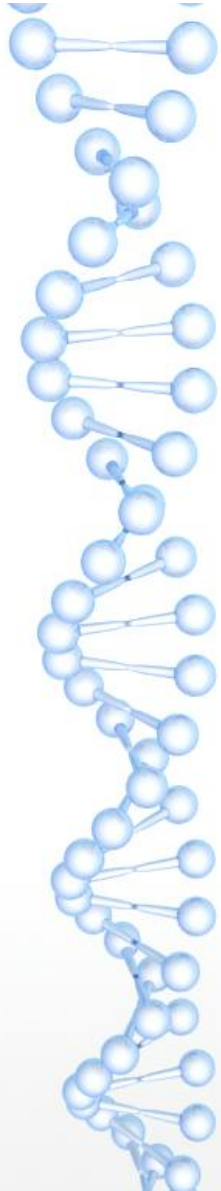
File-Transforming Commands

Many of Linux's text-manipulation commands are aimed at transforming the contents of files. These commands don't actually change files' contents, though; rather, they send the changed file to standard output. You can then pipe this output to another command or redirect it into a new file.

Converting Tabs to Spaces with *expand*

Sometimes text files contain tabs but programs that need to process the files don't cope well with tabs; you may want to convert tabs to spaces. The expand command does this.

By default, expand assumes a tab stop every eight characters. You can change this spacing with the -t *num* or --tabs=*num* option, where num is the tab spacing value.



Exploring Linux command-line tools

Processing Text Using Filters

File-Transforming Commands

Sorting Files with *sort*

Sometimes you'll create an output file that you want sorted. To do so, you can use a command that's called, appropriately enough, *sort*. This command can sort in several ways, including the following:

Ignore case Ordinarily, *sort* sorts by ASCII value, which differentiates between uppercase and lowercase letters. The -f or --ignore-case option causes sort to ignore case.

Month sort The -M or --month-sort option causes the program to sort by three-letter month abbreviation (JAN through DEC).

Numeric sort You can sort by number by using the -n or --numeric-sort option.

Exploring Linux command-line tools

Processing Text Using Filters

File-Transforming Commands

Sorting Files with *sort* (continued)

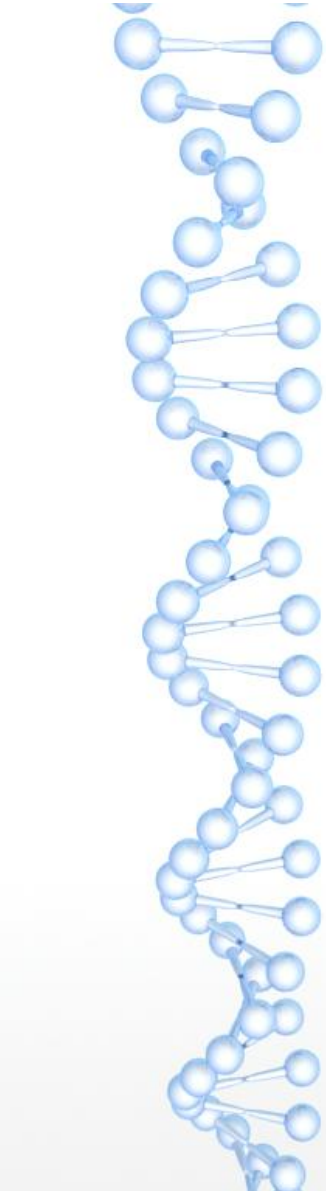
Reverse sort order The -r or --reverse option sorts in reverse order.

Sort field By default, sort uses the first field as its sort field. You can specify another field with the -k field or --key=field option. (The field can be two numbered fields separated by commas, to sort on multiple fields.)

As an example, suppose you wanted to sort listing1.1.txt by first name. You could do so like this:

```
$ sort -k 3 listing1.1.txt

555-2397 Beckett, Barry
555-5116 Carter, Gertrude
555-9871 Orwell, Samuel
555-7929 Jones, Theresa
```

The sort command supports a large number of additional options, many of them quite exotic. Consult sort's man page for details.



Exploring Linux command-line tools

Processing Text Using Filters

File-Viewing Commands

Sometimes you just want to view a file or part of a file. A few commands can help you accomplish this goal without loading the file into a full-fledged editor.

Viewing the Starts of Files with *head*

Sometimes all you need to do is see the first few lines of a file. This may be enough to identify what a mystery file is, for instance; or you may want to see the first few entries of a log file to determine when that file was started. You can accomplish this goal with the head command, which echoes the first 10 lines of one or more files to standard output.

You can modify the amount of information displayed by head in two ways:

Specify the number of bytes The -c num or --bytes=num option tells head to display num bytes from the file rather than the default 10 lines.

Specify the number of lines You can change the number of lines displayed with the -n num or --lines=num option.

Exploring Linux command-line tools

Processing Text Using Filters

File-Viewing Commands

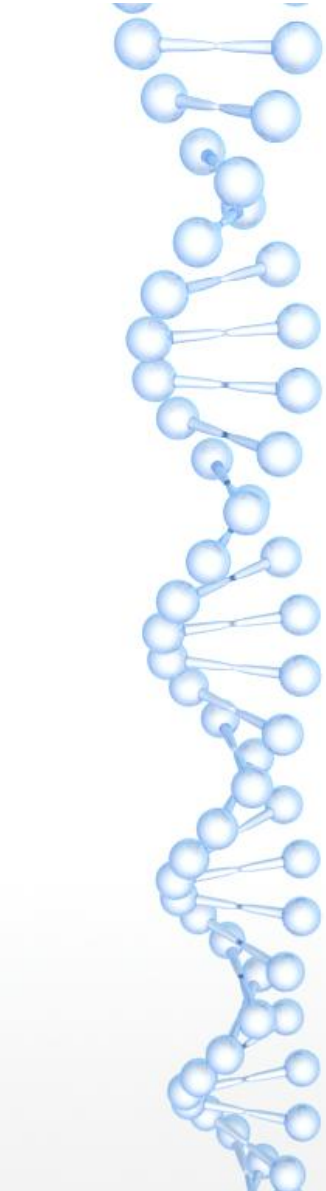
Viewing the Ends of Files with *tail*

The tail command works just like head, except that tail displays the last 10 lines of a file. (You can use the -c/--bytes and -n/--lines options to change the amount of data displayed, just as with head.) This command is useful for examining recent activity in log files or other files to which data may be appended.

The tail command supports several options that aren't present in head and that enable the program to handle additional duties, including the following:

Track a file The -f or --follow option tells tail to keep the file open and to display new lines as they're added.

Some additional options provide more obscure capabilities. Consult tail's man page for details.



Exploring Linux command-line tools

Processing Text Using Filters

File-Viewing Commands

Paging Through Files with *less*

The *less* command's name is a joke; it's a reference to the *more* command, which was an early file pager. The idea was to create a better version of *more*, so the developers called it *less*.

The idea behind *less* (and *more*, for that matter) is to enable you to read a file a screen at a time. When you type *less filename*, the program displays the first few lines of filename. You can then page back and forth through the file:

- Pressing the spacebar moves forward through the file one screen at a time.
- Pressing **B** moves backward through the file one screen at a time.
- Pressing **D** moves forward through the file half a screen at a time.
- Pressing **U** moves backward through the file half a screen at a time.
- The Up and Down arrow keys move up or down through the file one line at a time.

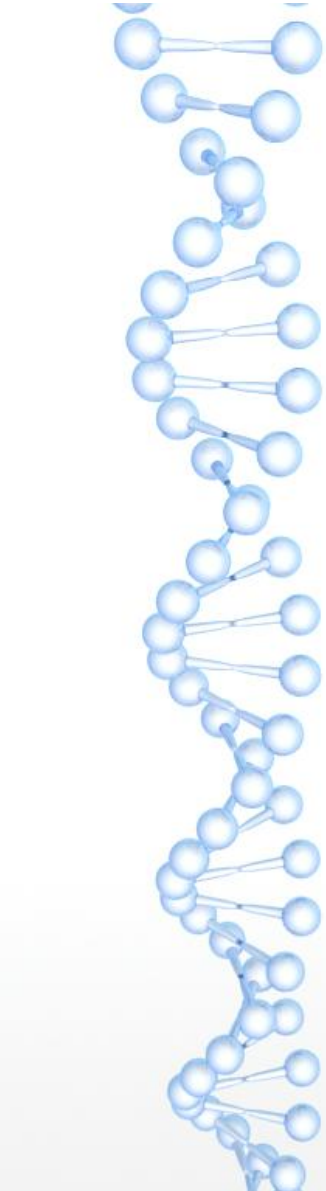
Exploring Linux command-line tools

Processing Text Using Filters

File-Viewing Commands

Paging Through Files with *less* (continued)

- You can search the file's contents by pressing the slash (/) key followed by the search term. Typing **n** alone repeats the search forward, while typing **N** alone repeats the search backward.
- You can move to a specific line by typing g followed by the line number, as in **50g** to go to line 50.
- You can move to an approximate percentage position of the file by typing g followed by the line number, as in **50p** to go to the 50% of the file.
- **g** will take you to the beginning of the file, while **G** will take you to the end of the file.
- When you're done, type **q** to exit from the program.



Exploring Linux command-line tools

Processing Text Using Filters

File-Viewing Commands

Extracting Text with *cut*

The cut command extracts portions of input lines and displays them on standard output. You can specify what to cut from input lines in several ways:

By byte The -b list or --bytes=list option cuts the specified list of bytes from the input file. (The format of a list is described shortly.)

By character The -c list or --characters=list option cuts the specified list of characters from the input file.

By field The -f list or --fields=list option cuts the specified list of fields from the input file. By default, a field is a tab-delimited section of a line, but you can change the delimiting character with the -d char, --delim=char, or --delimiter=char option option, where char is the character you want to use to delimit fields.

Exploring Linux command-line tools

Processing Text Using Filters

File-Viewing Commands

Extracting Text with *cut* (continued)

The cut command is frequently used in scripts to extract data from some other command's output. For instance, suppose you're writing a script and the script needs to know the hardware address of an Ethernet adapter. This information can be obtained from the ifconfig command:

```
$ ifconfig eno2
eno2: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.168.254  netmask 255.255.255.0  broadcast 192.168.168.255
        inet6 fe80::a6ba:dbff:fee1:4be7  prefixlen 64  scopeid 0x20<link>
        ether a4:ba:db:e1:4b:e7  txqueuelen 1000  (Ethernet)
        RX packets 23450459  bytes 2775486516 (2.5 GiB)
        RX errors 0  dropped 4  overruns 0  frame 0
        TX packets 2779622  bytes 365258758 (348.3 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 17
```

Exploring Linux command-line tools

Processing Text Using Filters

File-Viewing Commands

Extracting Text with *cut* (continued)

Unfortunately, most of this information is extraneous for the desired purpose. The hardware address is the 6-byte hexadecimal number following HWaddr. To extract that data, you can combine grep (described shortly, in “Using grep”) with cut in a pipe:

```
$ ifconfig eno2 | grep ether | cut -d" " -f10
a4:ba:db:e1:4b:e7
```

Of course, in a script you would probably assign this value to a variable or otherwise process it through additional pipes.

Exploring Linux command-line tools

Processing Text Using Filters

File-Viewing Commands

Obtaining a Word Count with *wc*

The *wc* command produces a word count (that's where it gets its name), as well as line and byte counts, for a file:

```
$ wc file.txt
308 2343 15534 file.txt
```

This file contains 308 lines (or, more precisely, 308 newline characters); 2,343 words; and 15,534 bytes. You can limit the output to the newline count, the word count, the byte count, or a character count with the --lines (-l), --words (□w), --bytes (-c), or --chars (-m) option, respectively. You can also learn the maximum line length with the --max-line-length (-L) option.

Exploring Linux command-line tools

Using Regular Expressions

Many Linux programs employ regular expressions, which are tools for expressing patterns in text. At their simplest, regular expressions can be plain text without adornment. Certain characters are used to denote patterns, though.

Understanding Regular Expressions

Two forms of regular expression are common: basic and extended.

Which form you must use depends on the program; some accept one form or the other.

The differences between basic and extended regular expressions are complex and subtle, but the fundamental principles of both are similar.

Exploring Linux command-line tools

Using Regular Expressions

Understanding Regular Expressions (continued)

The simplest type of regular expression is an alphabetic string, such as *Linux* or *HWaddr*. These regular expressions match any string of the same size or longer that contains the regular expression. For instance, the HWaddr regular expression matches HWaddr. The real strength of regular expressions comes in the use of non-alphabetic characters, which activate advanced matching rules:

Bracket expressions Characters enclosed in square brackets ([]) constitute bracket expressions, which match any one character within the brackets. For instance, the regular expression b[aeiou]g matches the words bag, beg, big, bog, and bug.

Range expressions A range expression is a variant on a bracket expression. Instead of listing every character that matches, range expressions list the start and end points separated by a dash (-), as in a[2-4]z. This regular expression matches a2z, a3z, and a4z.

Exploring Linux command-line tools

Using Regular Expressions

Understanding Regular Expressions (continued)

Any single character The dot (.) represents any single character except a newline. For instance, a.z matches a2z, abz, aQz, or any other three-character string that begins with a and ends with Z.

Start and end of line The carat (^) represents the start of a line, and the dollar sign ($) denotes the end of a line.

Repetition operators A full or partial regular expression may be followed by a special symbol to denote how many times a matching item must exist. Specifically, an asterisk (*) denotes zero or more occurrences, a plus sign (+) matches one or more occurrences, and a question mark (?) specifies zero or one match. The asterisk is often combined with the dot (as in .*) to specify a match with any substring. For instance, A.*Lincoln matches any string that contains A and Lincoln, in that order—Abe Lincoln and Abraham Lincoln are just two possible matches.

Exploring Linux command-line tools

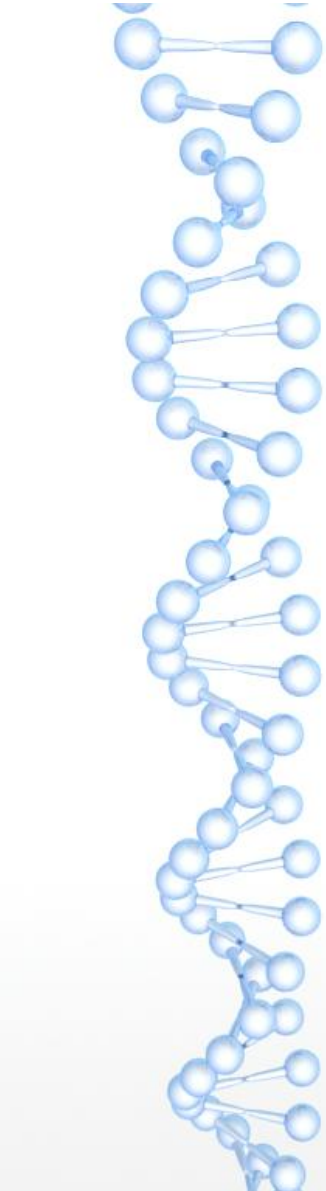
Using Regular Expressions

Understanding Regular Expressions (continued)

Multiple possible strings The vertical bar (|) separates two possible matches; for instance, car|truck matches either car or truck.

Parentheses Ordinary parentheses (()) surround subexpressions. Parentheses are often used to specify how operators are to be applied; for example, you can put parentheses around a group of words that are concatenated with the vertical bar, to ensure that the words are treated as a group, any one of which may match, without involving surrounding parts of the regular expression.

Escaping If you want to match one of the special characters, such as a dot, you must escape it—that is, precede it with a backslash (\). For instance, to match a computer hostname (say, twain.example.com), you must escape the dots, as in twain\.example\.com.



Exploring Linux command-line tools

Using Regular Expressions

Using *grep*

The grep command is extremely useful. It searches for files that contain a specified string and returns the name of the file and (if it's a text file) a line of context for that string. The basic grep syntax is as follows:

```
grep [options] regexp [files]
```

The regexp is a regular expression, as just described. The *grep* command supports a large number of options. Some of the more common options enable you to modify the way the program searches files.

Exploring Linux command-line tools

Using Regular Expressions

Using *grep* (continued)

Count matching lines Instead of displaying context lines, grep displays the number of lines that match the specified pattern if you use the -c or --count option.

Specify a pattern input file The -f file or --file=file option takes pattern input from the specified file rather than from the command line.

Ignore case You can perform a case-insensitive search, rather than the default case-sensitive search, by using the -i or --ignore-case option.

Search recursively The -r or --recursive option searches in the specified directory and all subdirectories rather than simply the specified directory. You can use rgrep rather than specify this option.

Exploring Linux command-line tools

Using Regular Expressions

Using *grep* (continued)

Use an extended regular expression The *grep* command interprets *regexp* as a basic regular expression by default. To use an extended regular expression, you can pass the -E or --extended-regexp option. Alternatively, you can call *egrep* rather than grep; this variant command uses extended regular expressions by default.

A simple example of grep uses a regular expression with no special components:

```
$ grep -r eno2 /etc/
```

This example finds all the files in /etc that contain the string eth0 (the identifier for the first Ethernet device). Because the example includes the -r option, it searches recursively, so files in subdirectories of /etc are examined as well as those in /etc itself. For each matching text file, the line that contains the string is printed.

Exploring Linux command-line tools

Using Regular Expressions

Using *grep* (continued)

Ramping up a bit, suppose you want to locate all the files in /etc that contain the string eno1 or eno2. You can enter the following command, which uses a bracket expression to specify both variant devices:

```
$ grep eno[12] /etc/*
```

A still more complex example searches all files in /etc that contain the hostname *twain. example.com* or *bronto.pangaea.edu* and, later on the same line, the number 127. This task requires using several of the regular expression features. Expressed using extended regular expression notation, the command looks like this:

```
$ grep -E “(twain\.example\.com|bronto\.pangaea\.edu).*127” /etc/*
```

Exploring Linux command-line tools

Using Regular Expressions

Using *grep* (continued)

This command illustrates another feature you may need to use: shell quoting. Because the shell uses certain characters, such as the vertical bar and the asterisk, for its own purposes, you must enclose certain regular expressions in quotes lest the shell attempt to parse the regular expression as shell commands.

You can use *grep* in conjunction with commands that produce a lot of output.

For example, suppose you want to find the process ID (PID) of a running xterm. You can use a pipe to send the result of a ps command through grep:

```
# ps aux | grep xterm
```

The result is a list of all running processes called *xterm*, along with their PIDs. You can even do this in series, using *grep* to further restrict the output on some other criterion, which can be useful if the initial pass still produces too much output.

Exploring Linux command-line tools

Using Regular Expressions

Using sed

The *sed* command directly modifies the contents of files, sending the changed file to standard output. Its syntax can take one of two forms:

```
sed [options] -f script-file [input-file]
sed [options] script-text [input-file]
```

In either case, *input-file* is the name of the file you want to modify. (Modifications are temporary unless you save them in some way).

The script (*script-text* or the contents of *script-file*) is the set of commands you want sed to perform. When you pass a script directly on the command line, the script-text is typically enclosed in single quote marks.

Exploring Linux command-line tools

Using Regular Expressions

Using sed (continued)

In operation, sed looks something like this:

```
$ sed ‘s/2021/2022/‘ cal-2021.txt > cal-2022.txt
```

This command processes the input file, cal-2021.txt, using sed’s s command to replace the first occurrence of 2021 on each line with 2022. (If a single line may have more than one instance of the search string, you must perform a global search by appending g to the command string, as in s/2021/2022/g.) By default, sed sends the modified file to standard output, so this example uses redirection to send the output to cal-2022.txt.

Exploring Linux command-line tools

Using find

The command **find** is used to search for files in the given directory, hierarchically starting at the parent directory and moving to sub-directories.

```
$ find /usr/sbin -name *.sh
/usr/sbin/duo_unix_support.sh
/usr/sbin/pm-utils-bugreport-info.sh
/usr/sbin/alsa-info.sh
/usr/sbin/alsabat-test.sh
```

The `-name‘ option makes the search case sensitive. You can use the `-iname‘ option to find something regardless of case. (* is a wildcard and searches all the file having extension ‘.sh‘ you can use filename or a part of file name to customize the output).

Exploring Linux command-line tools

Using ps

ps (Process) gives the status of running processes with a unique Id called PID.

```
$ ps

 PID TTY          TIME CMD
 4170 pts/1    00:00:00 bash
 9628 pts/1    00:00:00 ps
```

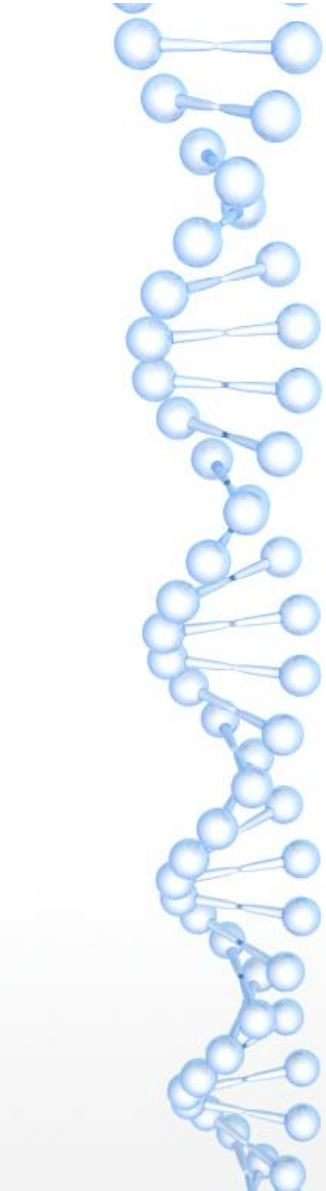
To list status of all the processes along with process id and PID, use option '-A'.

```
$ ps -A

 PID TTY          TIME CMD
    1 ?        00:00:01 init
    2 ?        00:00:00 kthreadd
....
    8 ?        00:00:00 migration/0
    9 ?        00:00:00 rcu_bh
```

This command is very useful when you want to know which processes are running or may need PID sometimes, for process to be killed. You can use it with 'grep' command to find customized output. For example,

```
$ ps -A | grep -i ssh

 1500 ?        00:09:58 sshd
 4317 ?        00:00:00 sshd
```

Here 'ps' is pipelined with 'grep' command to find customized and relevant output of our need.



UNIVERSITY OF TORONTO

Questions?

SciNet