Shell Command Notes

These are commonly-used Unix commands. Those covered in the *learn* tutorials have the lesson in italics under 'Notes' (the ordering does not follow *learn* exactly). Print this out and fill it in while doing tutorials, reviewing class notes, consulting manuals, or browsing the web. Most items have at least one blank to fill in.

A number in parentheses after a name, such as `date(1)` or `fgets(3)`, refers to a section of the on-line manual. To learn more about `time(2)`, you would run '`man 2 time`'. You should consult the manual to fill these pages in completely. (All the commands on these pages are in section 1.)

Things in brackets, [and], are optional. For example, '`cal [[month] year]`' means you can optionally give a year, or a month and a year, but not a month by itself. (If you only give one number, it's treated as a year.)
To test it out, run '`cal`', '`cal 1969`', '`cal 7 1969`', and then '`cal 7`'.

Because `l` and `1` may be confused, they will be labelled with (ell) or (one), but don't type that part.

1 Information

These commands give you information about what's going on.

Command	Flags	Arguments	Description	Notes
`date`		*(none)*	prints current date & time	*files 0.1b*
	`-R`		Use RFC 2822 format.	
	`+%s`		formats date as Unix time (# of seconds since 1970-01-01)	
	`'+%F %T'`		YYYY-MM-DD_hh:mm:ss	
`hostname`			prints hostname of the system	
`pwd`			prints full pathname current working directory	
`uptime`			current time, "up", system runtime, user count, system load avg	
`who`			show who is logged on	
`whoami`			returns username of current user	

2 Shell Wildcards for File Selection

These characters let you make patterns to specify multiple files without a lot of typing. Files whose names start with a period ('.') are ignored unless the period is specifically included in the first position. (Note: the square brackets in this section, `[` and `]`, don't indicate options, they mean to type square brackets.)

Character	Meaning	Notes
`?`	Any single character	*files 5.1a*
`*`	Any number (including zero) of any characters	*files* 6.1a
`[` *list* `]`	A set of particular characters in brackets	*files 7.1a*
`[` *range* `]`	A limited range of characters in brackets	*files 8.1a*

3 Working with Special Characters

To type a control character, such as control-H (notation ^H), you first type ^V (as in *verbose*). If you type **`grep ^V^H memo2.txt`**, it will run **`grep ^H memo2.txt`**, looking for `backspace` characters in the file `memo2.txt`. You can type a literal ^V by just typing it twice. (*morefiles* _____)

To use a string containing one or more shell wildcards as an argument to a program, you can use single quotes: **`grep 'when?' memo1.txt`**. The shell will treat '?' as a regular character. (*morefiles* _____)

@ → symbolic link / extended attributes
* → executable > → door
= → socket / → directory
| → named pipe

4 Working With Files

4.1 Examine

These commands show you things about your files; name, location, times, permissions.

Command	Flags	Arguments	Description	Notes
`ls`		[files]	list names of files that match w/ the argument	*files 0.1c, 4.1a*
	-1 (one)		list one file per line	
	-a		lists all files, including ones whose names start w/ '.'	
	-d		lists only directories	
*	-F		appends symbols to file names (useful info)	
	-O		omit group from long listing (simplifies column counting for awk/sort/&c.)	
	-i		include inode number	
	-l (ell)		long-format listing, w/ dates, sizes, & permissions	*morefiles 0.1a*
	-r		lists files in reverse order	
	-s		list files along w/ their size	
	-t		sort list by modification time (newest first)	*morefiles 0.1e*
`stat`		files	display file or file system status	

Command	Flags	Arguments	Description	Notes
`tree`		[folders]	lists contents of directories in a tree-like format	
	-a		show all files	
	-I	pattern	don't list files that match wildcard pattern	
	-L	number	lists tree to specified level (depth)	
	-P	pattern	lists files that match the wild card pattern	
	-p		print protections for each file (as per ls -l)	

patterns like "part*" for files starting w/ part

Command	Flags	Arguments	Description	Notes
`locate`		pattern(s)	find files by name	
	-b		match base name against specified patterns	
	-i		ignore case sensitivity & find matches for upper/lower case queries	

`find` works a little differently than the other commands in two ways: the directory names (if any) come first, instead of last, and (2) instead of options, it has predicates: that is, things which are true or false.

Command	Arguments	Predicates	Predicate is true if:
`find`	[dirnames]		examine every file in named directories and test files against predicate list
		-mtime [+-]#	file contents were modified # days ago
		-name 'name'	the file is named name
		-size [+-]#[c]	file is of size number disk blocks
		-user 'name'	file is owned by the user name

(+- → >< ; c → chars)

4.2 Management

These commands change files, or change your position in tree: create, remove, rename, relocate, set permissions.

Command	Flags	Arguments	Description	Notes
`cd`		[folder]	changes working directory to the one passed as argument	
		-	go back to last directory (don't name a folder '-')	
`mv`		file(s) destination	changes names of file to 2nd argument	*files 2.2a*
	-i		asks confirmation before overwriting existing file	
	-f		forcefully overwrites destination file & deletes source file	
`cp`		file(s) destination	creates a 2nd copy of a file (old file not removed)	*files 11.1a*
	-i		asks confirmation before overwriting destination file	
	-f		forcefully deletes destination file 1st before copying	
	-r		copies directory structure	
`rm`		files	removes file FOREVER (undoable)	*files 10.1a*
	-i		asks confirmation before removing file (press y)	
	-f		forcefully removes the file	
	-r		deletes all files/subdirectories of parent directory	**Caution!**
`touch`		files	creates file w/o any content (empty) w/ argument as filename	
`mkdir`		files	creates directory w/ argument as its name	
	-p		creates directory & any parents that don't already exist	
`rmdir`		files	removes directory w/ specified name	safe
	-p		removes all components in each directory in argument pathname	

`chmod` requires both a permission specification and one or more filenames.

Command	Flags	Arguments	Description	Notes
`chmod`		permission file(s)	change mode bits of named file/directory to specified permission setting	
			print a message for each file changed	
	`-R`	pattern		**Caution!**

Standard file permissions can be seen with `ls -l` or `stat`, and look something like this: `rwxr-xr-x`. These are three groups of three; the groups are for the **U**ser, **G**roup, and **O**thers. Each group refers to **R**ead, **W**rite, and e**X**ecute permission.

The permissions `rwxr-x---` for a directory mean that the user (first group of three, `rwx`) can read the directory (see what files are in it), write the directory (add or remove files), and execute programs in the directory (cd into it). Members of the user's group (second group of three, `r-x`) can read the directory and cd into it, but not add or remove any files. Others (third group of three, `---`) can't do anything at all.

To set permissions with `chmod`, you can specify all nine bits using octal (base-8) notation. `chmod 755 foobar` would take the octal as its binary value, `111101101`, and turn on the permissions with `1`, giving `rwxr-xr-x`. An octal permission of `644` would give `rw-r--r--`.

Mnemonic permissions can be used to turn on or off certain bits without affecting others; `chmod u+x prog1` would turn on the **U**ser's e**X**ecute permission without changing any other permissions. Using `go-rwx` would turn off **G**roup and **O**thers permission for all of **R**ead, **W**rite, and e**X**excute, without affecting the user's permissions.

5 File Content

5.1 Examination

These commands give you ways to see what's in your files.

Command	Flags	Arguments	Description	Notes
cat		[files]	print file(s)	*files 3.2a*
	-v		show non-printing character in output	
	-e		shows end of a line & any gap bet paragraphs w/ '$'	
	-t		show tab separated lines (tabspace represented by '^I'	
	-n		print the file but w/ line #s	*files 3.2c*
diff		file1 file2	compares 2 files, no output if same	*files 12.1a*
	-B		compares while ignoring changes where lines are all blank	
	-b		compares while ignoring changes in amount of white space	
	-c		differences shown w/ indicators on how to make them the sa	
	-i		compares 2 files while ignoring case	
	-u		same as context mode but less redundant	
	-w		compares while ignoring white space (all)	
tail		[files]	show the end of output or a file (10 lines)	*morefiles 4.1f*
	-number		shows number lines from the end of output or file	*morefiles 4.1g*
	-f		causes tail to follow file's progress as it's updated (to stop, hit ^C)	
head		[files]	shows beginning of output or a file (10 lines)	*morefiles 4.1h*
	-number		shows number lines from the beginning of output or file	*morefiles 4.1i*
wc		[files]	shows # of lines, # of words, # of bytes in file	*morefiles 4.1d*
	-l (ell)		shows # of lines in a file	
	-w		shows # of words in a file	
	-c		shows # of characters in a file (& bytes)	
file		file(s)	identifies a file's contents	
pr		[files]		*morefiles 3.1c*
	-t			*morefiles 3.1d2*
	-number			
spell		[files]		*morefiles 1.1a*

for multiple files, separated by a space

useful when suspecting CRLF ending lines

if different, prints the different lines

context mode

unified mode

more shows a file one screenful at a time, waiting for instructions with a prompt such as `--More--(15%)`. It is usually run without options; more important the commands you can type at the prompt:

Key	Effect	Key	Effect	Key	Effect
SPACE	down one page	d	down one-half page	h	get help
b	up one page	RETURN	down one line	q	quit

There are many more, listed in the manual (and by typing `h`); these are by far the most common.

6 Input/Output Redirection

6.1 I/O with Files

Many commands, if no filenames are given, take input from the keyboard (the default *standard input*) and print their output to your screen (the default *standard output*). The shell can, if you wish, use a file for standard input and/or standard output instead of your keyboard or screen.

To redirect **output**, use > (greater-than); `date > now.txt` will save the current time to the file `now.txt`, and nothing will appear on screen. **Caution:** if the file `now.txt` exists, it will be destroyed and replaced with a new file. (*morefiles 2.1a*)

To redirect **input**, use < (less-than); `tr A-Z a-z < Notes1.txt` will cause `tr` to read from `Notes1.txt` and print it to the screen in all-lowercase letters, instead of reading from the keyboard. (*morefiles 4.1a*)

You can do both at once, if you wish. `pr -t5 < names.txt > neatnames.txt` will use `pr` to format the names in five columns (it Terminal mode, that is, without headers) and send the output to `neatnames.txt` (destroying and replacing the file `neatnames.txt`, if it exists).

For a single command, you can specify the redirections in any order: `pr -t5 > neatnames.txt < names.txt` does the exact same thing.

6.2 Using Pipes

To create a **pipe**, use | (vertical bar): `cat names* | pr -t5` will cause the shell to run `cat` on all the files whose names start with `names`, and send that to `pr`, which will format them into five columns.

This can be combined with redirection to a file: `cat names* | pr -t5 > allneatnames.txt` will do the same, and the shell will send the result to `allneatnames.txt` (replacing the file with that name, if one exists). (*morefiles 5.1b*)

Note: pipes and file redirection are done by the shell; the programs you are running, such as `cat` and `pr`, don't know they aren't reading from your keyboard or printing to your screen. They take input from standard input (abbreviated `stdin`) and send output to standard output (abbreviated `stdout`). This makes writing programs that use pipes very easy for the programmer: you can just write your program to read from `stdin` and write to `stdout`, without worrying about pipes or redirection at all. The shell does all the work.

What's more, the standard functions to read and print in most programming languages use `stdin` and `stdout` by default, so you don't even have to specify that.

6.3 Using 'tee'

`tee` is a pipe fitting, which acts as an actual T pipe fitting on regular pipes: the output goes two places instead of just one.

`date | tee now.txt` will show the current date and time on standard output, and also save it in the file `now.txt`.

These can be chained together in long pipelines: `date | tee a | tee b | tee c | tee d`
will save the current time in files `a`, `b`, `c`, and `d`, and also print it to the screen. That's a somewhat frivolous example, but it illustrates what happens. A more useful example might be using `grep` to select lines of output for a file, and using `sed` or `awk` to process those same lines for redirection to another file.

7 Miscellaneous Useful Commands

These are some things that can come in handy, which you can look up in the manual. Ones marked with a dagger(†) are especially handy with shell scripts.

Command	Flags	Arguments	Description	Notes
cal		[[month] year]	show a calendar	Try Sept 1752
bc			simple calculator	
	-l (ell)			
history				
man				
	-k		keyword search	
reset				
units		[from to]		
nl				
	-ba			
rev				
tac				
echo				
look				
comm				
wget				
xargs				
zip				
unzip				
basename†				
	-s			
dirname†				

 Id: ShellCommands.tex,v 1.16 2020/12/17 13:24:17 kilroy Exp kilroy