THE UNIVERSITY of TEXAS

HEALTH SCIENCE CENTER AT HOUSTON

SCHOOL of HEALTH INFORMATION SCIENCES

Introduction to UNIX Part II

For students of HI 6327 “Biomolecular Modeling”

Willy Wriggers, Ph.D.
School of Health Information Sciences

http://biomachina.org/courses/modeling/03.html

Class Objectives

- introduction to student accounts
- basic background in UNIX structure and features
- getting started
- directory navigation and control
- file maintenance and display commands
- shells
- text processing
- resources for future projects

Editing Files with vi Line Editor

Three modes:

- Command mode (“beep mode”)
- Insert mode (“no beep mode”)
- Command line mode (“colon mode”)

Commands are generally case sensitive

vi Cursor Movements

arrow keys (depending on terminal)

h, j, k, l alternates for arrows

^F	forward one screen
^B	back one screen
^D	down half screen
^U	up half screen

…and many more… (**man vi**)

vi Deleting Text

dd delete current line

[n] dd delete *[n]* line(s)

[n] x delete *[n]* characters

[n] yy yank *[n]* line(s) to buffer

[n] yw yank *[n]* word(s) to buffer

vi Inserting Text

p	puts yanked or deleted text after cursor
P	puts yanked or deleted text before cursor
i	insert text before the cursor
a	append text after the cursor
I	insert text at beginning of line
A	append text at end of line
o	open new line after current line
O	open new line before current line

vi Saving and Exiting

Esc	to reach colon mode
:w	write changes to file
:wq	write changes and quit
:w!	force overwrite of file
:q	quit if no changes made
:q!	quit without saving changes

Editing Files with emacs

- Visual editor
- Move around freely in buffer
- Self-explanatory menu system
- Customizable with .emacs configuration file
- Online help system ^**H** (control-H)
- Undo changes with ^_ (control-underscore)
- Windows version exists (see online course notes)

emacs Cursor Movements

arrow keys (most terminals)

Alternates:

^F	next character
^B	previous character
^N	next line
^P	previous line

emacs Cutting and Pasting Text

- Delete characters with **Delete** key
- Highlight regions of text with mouse (click and drag), or with ^**space** on either side
- Cut / delete a larger piece of text with ^**W** (wipe)
- Paste with ^**Y** (yank)

emacs Saving and Exiting

^X^S save buffer to file

^X^C exit emacs

Text Processing Commands

• grep / egrep / fgrep - search the argument for all occurences of the search string; list them

• awk / gawk - scan for patterns in a file and process the results

• sed - stream editor for editing files from script or command line

The search strings can be generalized to “**regular expressions**”

Regular Expressions

- allow pattern matching on text
- combine normal and special characters (metacharacters)
- should not be confused with wildcards for matching files

Regular Expression Syntax

Regular expressions come in three different forms:

- Anchors —tie the pattern to a location on the line
- Character sets —match a single character at a single position
- Modifiers —specify how many times to repeat the previous expression

Regular Expressions can be combined to form longer regular expressions.

Regular Expression Syntax

.	match any single character except newline
*****	match zero or more instances of single expression preceding it
[abc]	match any of the characters enclosed
[a-d]	match any character in enclosed range

Regular Expression Syntax

[^abc]	match any character NOT in the enclosed set
^exp	regular expression must start at the beginning of the line
exp$	regular expression must end at the end of the line
****	treat the next character literally

grep

grep *[options] regexp [files...]*

The grep utility is used to search for regular expressions in UNIX files.

fgrep searches for exact strings. egrep uses “extended” regular expressions.

Some options for grep are:

-i ignore case

-v display only lines that dont match

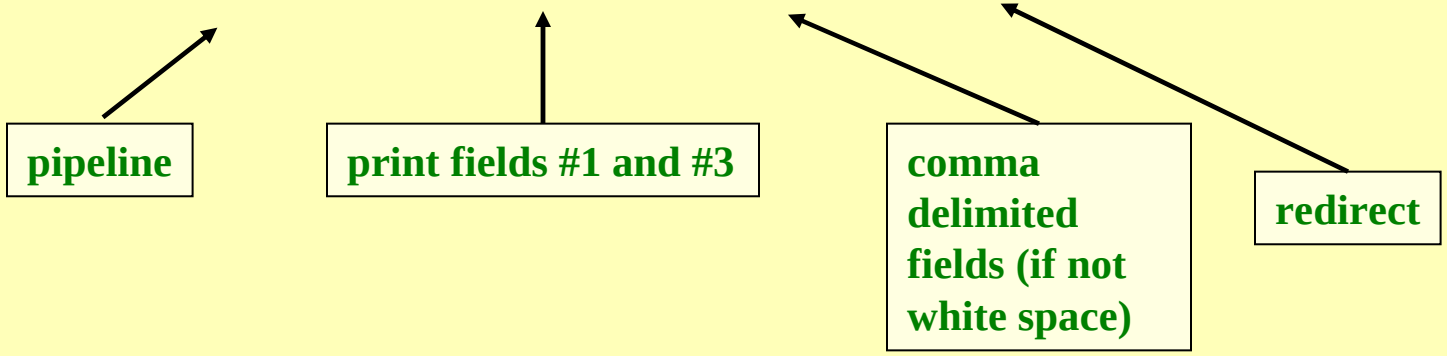
-n display line number with the line where match was found

gawk

gawk is the open source implementation of the awk pattern scanning and text processing language. Awk is a full-featured programming language, beyond the scope of class (see O'Reilly book "sed & awk").

Often, gawk is used to extract fields from data files:

cat file1 | gawk '{print $1 "," $3 }' > file2



sed

sed *[options] '{script}' [file]*

The sed utility is a stream editor. A stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline).

Most often, sed is used to replace strings:

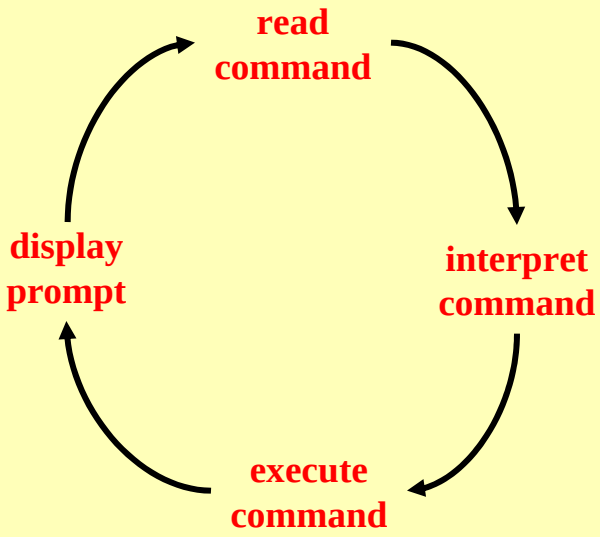
cat file1 | sed 's/regexp/replacement/g' > file2

Questions: Text and String Editing

The Shell

The shell sits between you and the OS

- acts as a command interpreter
- reads input
- translates commands into actions



Bourne Shell (sh)

- good features for I/O control - often used for scripts
- awkward ‘Old School’ syntax
- not well suited for interactive users
- default prompt is `$`

C Shell (csh)

- uses C-like syntax for scripting
- I/O more awkward than Bourne shell
- somewhat nicer for interactive use
- job control
- history
- default prompt is %
- uses ~ symbol to indicate a home directory (user's or others')

T-C Shell (tcsh)

- backward compatible with C-shell commands and scripts
- enhanced for interactive use
- file name completion
- command line editing (with left/right arrows)
- command history (with up/down arrows)

Bourne Again Shell (bash)

- backward compatible with Bourne shell commands and scripts
- has some features from C-shell as well (~, history)
- enhanced for interactive use
- file name completion
- command line editing (with left/right arrows)
- command history (with up/down arrows)

Which Shell to Use?

- bash or tcsh: easy to fix typos or redo previous commands.
- most built-in commands are nowadays available in both
- personal programming preference and history
- easy to switch between shells:
- e.g. Cygwin and Linux have bash by default, but by calling tcsh in .profile (bash startup file) we can start with tcsh terminal

Environment Variables

DISPLAY

EDITOR

PATH

TERM …

(list with **env**)

csh/tcsh **setenv NAME *value***

sh/bash **NAME=*value*; export NAME**

Shell Variables

PS1 (sh/bash)

prompt (csh/tcsh)

others as needed (list with **set**)

csh/tcsh **set name=*value***

sh/bash **name=*value***

These are used by the shell and shell scripts; not seen or used by external programs

Shell Startup

The files .profile (sh) or .bash_profile (bash) or .login (csh/tcsh) are used at login to:

- set path (search path for executables)
- define functions
- set terminal parameters (stty)
- set terminal type
- set default file permissions (umask)

Sample .profile or .bash_profile

```
PATH=/usr/bin:/usr/ucb:/usr/local/bin:.
export PATH
PS1="{ `hostname` `whoami` } "
ls() { /bin/ls -sbF "$@"; }
ll() { ls -al "$@"; }
stty erase ^H
eval `tset -Q -s -m ':?xterm'`
umask 077
```

Startup Files sh/csh Family Shells

File	Shell	Comments
.login	csh, tcsh	Read only for login shells
.cshrc	csh, tcsh	Read for each new shell
.tcshrc	tcsh	Tcsh ignores .cshrc if .tcshrc exists
.logout	csh, tcsh	Executed at logout
.profile	sh, bash	
.bash_profile	bash	Bash ignores .profile if .bash_profile exists.
.bash_logout	bash	Executed at logout.
.bashrc	bash	Used if shell reads /etc/bashrc if it exists

Customizing Start-Up Files

- Files are placed in your home directory
- Define your favorite prompt, shell parameters and more
- Much information is available online:

  Search for “customize bash” at google

Shell History

tcsh and bash retain information about former commands executed within the shell

Customize variables to set number of commands retained:

E.g. in .profile:

HISTSIZE=1000

History saved in ~/.history or ~/.bash_history between logins.

History Shortcuts

`$` **history** ***nn***

prints last *nn* commands

`$` **!!**

repeats the last command

`$` **!*nn***

repeats the command numbered *nn*

`$` **!*string***

repeats latest command starting with *string*

Input and Output

I/O redirection and piping in UNIX:

- output redirection to a file
- input redirection from a file
- piping (pipelining): output of one command becomes the input of a subsequent command

Standard File Descriptors

stdin Standard input to the program

stdout Standard output from the program

stderr Standard error output

These are not called by name at shell prompt, but are often referenced by these names.

File Descriptors

stdin normally from the keyboard, but can redirect from a file or command

stdout & **stderr** normally to the terminal screen, but can redirect either or both to a file or command

I/O Redirection

> redirect standard output to file

command > outfile

>> append standard output to file

command >> outfile

< input redirection from file

command < infile

| pipe output to another command

command1 | command2

csh/tcsh Advanced Redirection

>& file redirect stdout and stderr to *file*

>>& file append stdout and stderr to *file*

|& command pipe stdout and stderr to *command*

To redirect stdout and stderr to separate files:

% **(*command* > *outfile*) >& *errfile***

sh/bash Advanced Redirection

2>file	direct stderr to *file*
>file 2>&1	direct both stdout and stderr to *file*
>>file 2>&1	append both stdout and stderr to *file*
2>&1 | command	pipe stdout and stderr to *command*

To redirect stdout and stderr to two separate files:

$ **command > outfile 2 > errfile**

To discard stderr:

$ **command 2 > /dev/null** (/dev/null: UNIX “black hole”)

Special Command Symbols

;	command separator
&	run the command in the background
&&	run the following command only if previous command completes successfully
||	run the following command only if previous command did not complete successfully
( )	grouping — commands within parentheses are executed in a subshell

Quoting

****	escape the following character (take it literally)
'…'	don't allow any special meaning to characters within single quotes (except ! in csh)
"…"	allow variable and command substitution inside double quotes (does not disable $ and \ within the string)
`…`	take the output of command in backquotes and substitute it into the command line (works inside double-quotes)

Shell Scripts

- Similar to DOS batch files
- Quick and simple programming
- Text file, interpreted by shell, effectively new command
- List of shell commands to be run sequentially
- Set **execute permission**, no special extension necessary
- Best way to learn: Look at examples

First Line

Include full path to interpreter (shell)

#!/bin/sh

#!/bin/csh -f

- csh / tcsh follows C syntax, see man pages, books, and WWW
- In following only brief intro to sh/bash

Special sh/bash Variables

$# Number of arguments on command line

$0 Name that script was called as

$1 – $9 Command line arguments

$@ All arguments (separately quoted)

$* All arguments

$? Numeric result code of previous command

$$ Process ID of this running script

I/O (sh/bash)

echo ***output text***

Talk to user (or ask questions)

read ***variable***

Get input from user, put it in variable

Control Flow (sh/bash)

- test and []
- if [. . .]; then

  . . .

  fi
- case $*variable* in . . . esac
- for *variable* in . . .
- do . . . done

Check sh/bash man page for details, also look at examples.

Questions: Shells and Shell Programming

Resources

UNIX man pages

WWW:
http://www.utexas.edu/cc/docs/ccug1
http://www.ee.surrey.ac.uk/Teaching/Unix
http://www.ee.surrey.ac.uk/Docs/Unixhelp

O'Reilly UNIX and Linux Books:
http://unix.oreilly.com

Figure and Text Credits

Text and figures for this lecture were adapted in part from the following source, in agreement with the listed copyright statements:

http://wks.uts.ohio-state.edu/unix_course