

## Unix Basics

Tom Nichols

March 21, 2001

1

## Goals

---

- Improve your efficiency by...
  - Like `sed` and `awk`
- Using essential unix programs
  - Shorten commands with aliases
  - Changing defaults with environment variables
- Modifying your environment
  - Shorten commands with aliases
  - Changing defaults with environment variables
- Writing basic shell scripts
  - Automate repetitive tasks

2

## Outline

---

- Shell Basics I
- Basic UNIX Programs
- Shell Basics II
- Essential UNIX Programs
- Scripting

3

## Shell Basics I

---

- The Shell
  - Just another program
  - Accepts your keystrokes
  - Runs programs on your behalf
- More than one!
  - Everything here is about the C-shell (`csh`)
  - Others include
    - Bourne shell (`sh`)
    - T-shell (`tcsh`)
    - Korn-shell (`ksh`)

4

## Shell Basics I

---

- Most basic shell commands
  - `cd` - Change directory
  - Ex `cd ~`  
*Change to home directory*
  - `pwd` - Print working directory
  - `echo` - Print some text
  - Ex `echo Hello world`
- Most basic non-shell command
  - List contents of directory `ls`

5

## Shell Basics I: Wild Cards

---

- Special characters are “expanded” by the shell...
- `*`
  - Matches zero or more characters (any)
  - Ex `ls *foo`  
*Lists all files ending in foo, including `yofoo` & `foo`*
- `?`
  - Matches exactly one character (any)
  - Ex `ls foo-?ar`  
*Lists files such as `foo-bar` & `foo-har`  
but not `foo-ar`*
  - Ex `ls foo.???`  
*Lists files with three character suffix*

6

## Shell Basics I: Wild Cards

- [ ]
    - Matches exactly one character of those within brackets
- ```
Ex ls [abc]*
```
- Lists all files starting with letter a, b or c*

```
Ex ls *[A-Z]
```

*Lists all files ending with a capital letter*

  - [ ^ ]
    - Matches exactly one character, all *but* those within brackets

```
Ex ls *[^0-9]
```

*Lists all files not ending with a number*

7

## Shell Basics I: Wild Cards

- { , , }
- Matches any of strings listed
- ```
Ex ls *.{ps,pdf}
```
- Lists all files ending in either .ps or .pdf*

```
Ex echo {main,sub1,sub2}.{c,h}
```

*Prints main.c main.h sub1.c sub1.h  
sub2.c sub2.h*

```
Ex echo {main,sub{1,2}}.{c,h}
```

*Prints same thing... you can nest!*

8

## Shell Basics I: Variables - Shell variables

- You can define variables in the shell
  - Much like in Splus or any language.
  - Unlike Splus, you must mark a variable with a special character... \$
- Define with *set*, evaluate with \$

```
Ex set food = doughnuts
```

```
echo Yummie $food
```

*Sets variable food to have value "doughnuts"  
then prints "Yummie doughnuts"*

9

## Shell Basics I: Variables - Shell variables

- Only can set to a single "whitespace separated token"

```
Ex set food = doughnuts are good food
```

```
echo Yummie $food
```

*Creates an error or just prints "Yummie doughnuts"*

```
Ex set food = "doughnuts are good food"
```

```
echo Yummie $food
```

*Prints "Yummie doughnuts are good food"*

10

## Shell Basics I: Variables - Shell variables

- Some variables are used by the shell
    - home, user, prompt
- ```
Ex echo I am $user and I live at $home
```
- Prints I am nichols and I live at  
/afs/sph.umich.edu/usr/n/nichols*

```
Ex set prompt = YourWishIsMyShellCommand
```

*The shell will prompt you with that corny string  
instead of ">"*

11

## Shell Basics I: Variables - Shell variables

- You can set variables to save on typing

```
Ex set list = "prog.c dood.c blah.c  
prog.h"
```

```
ls -l $list
```

```
cp $list /tmp
```

```
lpr $list
```

12

## Shell Basics I: Variables - Environmental variables

---

- Almost identical to shell variables, but they
  - Use different definition command, `setenv`
  - Persist, they get passed to programs you call
    - Shell variables are *local*
    - Environmental variables are *global*
- By convention, always in all CAPITAL LETTERS
- Used by system to control defaults  
Ex `setenv PRINTER student_lp`  
`matlab`  
*This sets the default printer; Matlab will know about it*

13

## Shell Basics I: Command Substitution

---

- Using `!` (bang) you can save typing
- `!!` - The last command
  - `!! : n` - The *n*th word of the last command  
*Note n = 0 is the command name*
- `!!$` or `!$` - The last word of the last command
- `!!*` or `!*` - All of the words (except command name)

14

## Shell Basics I: Command Substitution

---

- Examples
  - `echo This is a really long and annoying command`  
*Prints "This is a really long and annoying command"*
  - `!!`  
*Prints it again*
  - `echo Just !!:3 !$`  
*Prints "Just a command"*

15

## Shell Basics I: Quiz

---

- `echo This is just a {test,quiz}`  
\_\_\_\_\_
- `set list = "!!:1 !!:0"`  
\_\_\_\_\_
- `echo !$`  
\_\_\_\_\_

16

## Getting the most out of UNIX

Tom Nichols

April 11, 2001

17

## Review: Shell Basics

---

- Shell
  - Just another program
  - But *also* a scripting language
- Shell Wildcards
  - `ls ?[a-z]*.{pdf,ps}`
- Shell Variables
  - `set food = "bagels are good food"`  
`echo Surely, $food`
- Shell Environmental Variables
  - `setenv FOOD "bagels are good food"`
  - `setenv PRINTER student_lp`
  - Unlike shell variables, environmental variables are inherited by new programs you start

18

## Basic Programs

---

- `cat` - Concatenate files  
Ex `cat out1 out2 out3`  
*Spews contents of the three files*

19

## Basic Programs

---

- `ls` - List files
    - Key options
      - `-l` Long listing
      - `-a` Include files beginning with `.` (dot)
      - `-t` Order by time of last modification (best w/ `-l`)
      - `-d` Do not list subdirectory contents
    - Key detail
    - If you give `ls` a directory name, it will list the *contents*
- Ex `ls`  
*Show files and subdirectories of current directory*
- Ex `ls *`  
*Same, but ALSO shows contents any subdirectories; `-d` option will suppress this behaviour*

20

## Basic Programs

---

- `more` or `less` - View one or more files
  - Key key commands
    - (space) Move forward one page
    - (return) Move forward one line
    - `q` quit
    - `b` Move backwards one page (better w/ `less`)
    - `/pattern` Search forward for pattern
    - `?pattern` Search backwards for pattern
    - `:n` If multiple files, skip to next file
    - `:p` If multiple files, go back to previous file

21

## Basic Programs

---

- `head` - View the first few lines of a file
  - Key options
    - `-n` Show first *n* lines (10 by default)
- `tail` - View the last few lines of a file
  - Key options
    - `-n` Show last *n* lines (10 by default)
    - `-f` Continually check the file; if it grows, show the new lines

22

## Basic Programs

---

- `man` - Get "help" on commands
  - Key options
    - `-k` Give single line summary of relevant commands
    - `-F` Sometimes needed w/ SPH commands
  - Key key commands: Same as `more`
- `sort` - Sort a file, one line at a time
  - Key options
    - `-r` Reverse the order of sort
    - `-n` Try to sort numbers correctly (2 before 10)

23

## Basic Programs

---

- `du` - Disk usage, directory sizes in KB
    - Key options
      - `-s` Silent, only report one number
- Ex `du 650`
- ```
480    650/tmp
5      650/faraway/R
295    650/faraway/data
303    650/faraway
2707   650
```
- Ex `du -s 650`
- ```
2707   650
```

24

## Basic Programs: Quiz

---

- Preview: Pipes
  - A way to connect commands
- Ex These two commands are about the same

```
more out1 out2 out3
cat out1 out2 out3 | more
```
- The | is the “pipe”
  - Connect the output of a command to the input of another

25

## Basic Programs: Quiz

---

- `ls -lat | head`  
\_\_\_\_\_
- `du -s * | sort -nr | head`  
\_\_\_\_\_
- `du -s */. | sort -nr | head`  
\_\_\_\_\_

26

## Shell Basics II: .cshrc/.login

---

- Any shell settings we discussed so far are not *persistant*
  - If you set the `prompt` variable and logout, when you log back in it will be gone
- `.login` & `.cshrc`
  - Shell *scripts* executed every time you `login (.login)` or start a new shell (`.cshrc`)
  - Should only have to put things in `.login`
  - But to keep life simple, put things in `.cshrc`
- To permantly change the prompt
  - Add this to the end of `.cshrc`

```
set prompt =
>YourWishIsMyShellCommand: "
```
  - Note the quotes; allows the extra space

27

## Shell Basics II: I/O redirection & pipes

---

- Every UNIX program has one input and two outputs
  - *stdin* Standard in - What you type
  - *stdout* Standard out - What you see
  - *stderr* Standard error - Error messages you see
- | Pipes
  - Connect programs *stdin* & *stdout*'s
- Ex `prog1 | prog2`
  - Sends *stdout* of `prog1` to *stdin* of `prog2`

28

## Shell Basics II: I/O redirection & pipes

---

- `< filename` - Input redirect
  - Feed file into program as if you typed it
- Ex `Splus < Ccmds.S`
  - As if you typed in the commands in Ccmds.S*
  - (There are better ways to do this w/ Splus)*

- `> filename` - Standard Output redirect
  - Save program output (*stdout*) to file
- Ex `ls -l > filelisting`
  - Saves long directory listing in filelisting*
- Ex `Splus < Ccmds.S > Ccmds.log`
  - Saves output in Ccmds.log*

29

## Shell Basics II: I/O redirection & pipes

---

- `>& filename` - Output redirect (*stdout and stderr*)
  - Ex `Splus < Ccmds.S >& Ccmds.log`
    - Saves normal and error output in Ccmds.log*
    - For example, if Splus cannot be found, that error will now go to Ccmds.log instead of the terminal.*
- `>> filename` and `>>& filename` - Appending redirect
  - Same as above, but `filename` is appended to, instead of overwritten.

30

## Shell Basics II: "Type out" stdin

- `prog << EOF ... EOF` - Specify stdin

```
Ex Splus << EOF
    u1 <- runif(100)
    median(u1)
    q()
EOF
```

- This will run Splus, find the median of 100 random uniforms, then quit
- Kinda dumb here, but *very* useful in scripts
- String doesn't have to be "EOF"... can be anything

31

## Shell Basics II: Process Control

- Through a single terminal, can control many processes
  - Normally, a program takes over your terminal
    - You can't do anything until it finishes
  - However, you can *background* a program if...
    - It doesn't take keyboard input
    - It doesn't write to the terminal
  - This is most useful under X windows
- Ex `netscape &`
- This puts Netscape into the background so you can continue to use the terminal.*

32

## Shell Basics II: Process Control

- Almost any program can satisfy this if you redirect the I/O
- ```
Ex Splus < Cnds.S >& Cnds.log &
```
- Splus normally reads from the keyboard and writes to the terminal, but we have redirected the I/O.*
- You don't have to use "&"
    - If you forget to background a job you can "stop" it and then "bg" it

33

## Shell Basics II: Process Control

- Background example
- ```
Ex netscape      oops... forgot to &-it      Press Ctrl-Z
    bg              This will put netscape into background
```
- Use `jobs` or `ps` to see the processes you started
  - Use `kill -9` to kill runaway jobs
- ```
Ex ps
    PID TTY          TIME CMD
    8433 pts/2        0:00 csh
    8842 pts/2        0:01 netscape
```
- ```
kill -9 8842          Brutally kills netscape
```
- With an editor (e.g. pine or emacs), always try `kill -HUP` first, so that it can try to save your work.

34

## Shell Basics II: Quote Flavors

- Three types of quotes
  - " (double), ' (single), and \ (backwards)
  - Single and double
    - Control how shell interprets spaces and variables
  - Backwards
    - Very special – runs a command

35

## Shell Basics II: Quote Flavors

- Double (")
    - Protects spaces and special characters, but still allows variables
- ```
Ex echo "Some      long space" some      short
    Prints Some      long space some short
```
- ```
Ex echo "Hey $user, special chars: | { * ?"
    Prints Hey nichols, special chars: | { * ?
```

36

## Shell Basics II: Quote Flavors

---

- Single (')
- Iron-clad protection of special characters and spaces
- Ex `echo This is a dollar sign: $` *Error!*
- Ex `echo This is a dollar sign: '$'` *OK*
- Ex `echo '$user is $user'`  
*Prints \$user is nichols*

37

## Shell Basics II: Quote Flavors

---

- Backwards (')
- Runs a command, and puts the output in its place
- Ex `echo Date and time now: `date` AD`  
*Prints:*  
Date and time now: Wed May 9 11:29:03 EDT 2001 AD
- We'll see how this is very useful with scripts

38

## Shell Basics II: Quiz 1

---

- `echo "Will $user cause '$user trouble'"`  
\_\_\_\_\_
- `Spplus < Ccmds.S >& Ccmds.log & tail -f Ccmds.log`  
\_\_\_\_\_

39

## Shell Basics II: Aliases

---

- Aliases most powerful aspect of C-shell
- Allows you replace long, complicated commands with shorthands
- Syntax  
→ `alias aliasname aliascommand`
- Common aliases  
`alias rm rm -i`  
*Every time you type `rm, rm -i` is actually used; asks if you're sure about deleting files*  
`alias cdthere cd`  
`/some/really/long/annoying/path/name`  
*Just like typing the command.*

40

## Shell Basics II: Aliases

---

- My favorite aliases  
→ `alias lsh "ls -lat | head"`  
*Shows the 10 most recently modified files. Note that I have to use quotes because of the pipe.*
- But what if I want to see the 10 most recent \*.c files?
  - `lsh *.c` is the same as typing  
`ls -lat | head *.c`
  - Need to have control over arguments...

41

## Shell Basics II: Aliases

---

- Controlling the arguments in aliases  
`alias lsh "ls -lat \!* | head"`  
*This is the same, except any arguments I type will go where the \!\* is.*
- The bang is special, in that it ignores single or double quotes; to protect it you have to *escape* it, with a \

42

## Shell Basics II: Aliases

---

- My favorite aliases
  - `alias lsm "ls -lat \!* | more"`  
*Shows the most recently modified files, as piped through head*
  - `alias psg "ps -elf | grep \!*"`  
*Shows all processes on machine, piped through grep.*

43

Unix Basics

Tom Nichols

May 9, 2001

44

## Shell Basics II: Aliases quiz

---

- `alias lsh "ls -lt \!* | head"`  
\_\_\_\_\_
- `alias psm "/usr/ucb/ps axuw | more"`  
\_\_\_\_\_

45

## Shell Scripting I: Basics

---

- A shell script is just a file filled with shell commands
- For any such file, can always just “source”  
Ex `source .cshrc`
  - This re-reads your shell startup file
- But to make an *executable script* need two things
  - First line must be  
`#!/bin/csh`
  - And the file must be given execution rights  
`chmod +x scriptname`
- An executable file can then be run just by typing the name

46

## Shell Scripting I: Variables

---

- Recall how to set a variable  
Ex `set food = "doughnuts and coffee"`  
Ex `echo I need $food`  
*Will print “I need doughnuts and coffee”*
- Can also set a “list” variable  
Ex `set food = ( "doughnuts" "and" "coffee" )`
  - Use square brackets [] to access each elementEx `echo I need $food[2] or $food[1]`  
*Prints I need coffee or doughnuts*

47

## Shell Scripting I: Variables

---

- You can access program arguments via list variables  
Ex `myscript this that the other thing`  
*This script has 5 arguments*
- Every shell program has `$argc` and `$argv`
  - `$argc` The number of arguments
  - `$argv` A list variable of the arguments
- Because these are *so* useful, they have a short cut
  - `$1` is `$argv[1]`
  - `$2` is `$argv[2]`
  - *etc* upto 9
  - `$*` is the whole list of args, `$argv`

48

## Shell Scripting I: Variables

- For example, consider `myscript`

```
#!/bin/csh
# My first shell script
echo Hello world
echo All args: $argv
echo First arg: $1

→ If I run
    myscript thisone that1
→ I get
    Hello world
    All args:  thisone that1
    First arg:  that1
```

49

## Shell Scripting I: Numeric Variables

- The shell has support for integer variables
- There is a special form of “set”, “@”
- One useful “@” operation is borrowed from C

```
Ex @ x = 3
Ex @ x = $x + 2
Ex echo $x Prints "5"

Ex @ x = 3
Ex @ x++
Ex echo $x Prints "4"
Ex @ x--
Ex echo $x Prints "3"
```

50

## Shell Scripting I: foreach loops

- Most useful shell script construct

```
→ foreach VarNam (list)
    commands
end
→ Loops through the values in a list
```

- Almost all of my shell scripts have this form

```
Ex To change the name of a bunch of files
#!/bin/csh
foreach f ($*)
    mv $f OLD$f
end
→ For every filename I specify, its name is prepended
with “OLD”
```

51

## Shell Scripting I: while loops

- “while” loops repeat until a statement is false

```
Ex To make 10 numbered copies of a file...
#!/bin/csh
set file = "stuff"
@ i = 1
while ( $i <= 10 )
    cp $file ${file}_${i}
    @ i++
end
→ Note the use of brackets {} to protect the variable name
– “$file_$1” would have caused an error, since
I don’t have a $file_ variable
→ Note that instead of “set file = "stuff"”
I could have used “set file = $1”
using a file specified as an argument
```

52

## Power Programs: grep

- Grep!

```
→ A single reason for using UNIX
→ Reads stdin or a file, only shows lines that match a string
→ Key options
– -i Ignore case
– -v Show all lines that do not match string
```

```
Ex grep nichols /etc/passwd
Shows my entry in the password table:
```

```
nichols:x:137710:10:Thomas E. Nichols:/afs/.../nichols:/usr/bin/csh
```

53

## Power Programs: grep

```
Ex grep Nichols /etc/passwd
Shows nothing
Ex grep -i Nichols /etc/passwd
Shows same as first example; case ignored
Ex grep -v nichols /etc/passwd
Shows everyone else’s entry
```

54

## Power Programs: sed

- sed - Stream Editor
  - A powerful programming language
  - But just one part is all you need for today
- sed s/string1/string2/
  - This will take a file (or a pipe) and change string1 to string2.
  - By default, it only changes the first occurrence each line
- sed s/string1/string2/g
  - Will change every occurrence
- Any character can be used instead of /  
Ex s@string1@string2@ is equivalent to above
  - This is very useful if your string uses /s

55

## Power Programs: sed examples

- A file-wide change
  - I need to change all references to Normal to Gaussian
  - Ex sed s/Normal/Gaussian/ paper.tex > paper2.tex
- In a backwards-quote
  - Ex set file = ravol\_e40.02.03.101.0480.img
  - mv \$file `echo \$file | sed s/\_e40.02.03.101//`
  - This is the same as
  - mv ravol\_e40.02.03.101.0480.img ravol.0480.img
- Useful for dealing with PC/UNIX filename problems  
Ex sed 's/GIF/gif/' PCfilenames > UNIXfilenames

56

## Power Programs: awk

- awk - Named after its creators  
Aho, Weinberger & Kernighan
- Powerful tool for dealing with formatted text
- Again, a full programming language, but only need a few bits
- Fundamental awk
  - Each line read is divided into records
  - Each record is a number-named variable: \$1, \$2, \$3
    - Like the shell!
  - By default, records are whitespace-separated words
  - But other separators (e.g. commas) can be used
  - The awk “program” must be surrounded by {}'s

57

## Power Programs: awk examples

- I want to see the list of file modification dates
  - ls -l

```
-rw-r--r-- 1 nichols staff      8 Apr 11 11:34 UnixBrownBag.aux
-rw-r--r-- 1 nichols staff 28664 Apr 11 11:34 UnixBrownBag.dvi
-rw-r--r-- 1 nichols staff  8225 Apr 11 11:34 UnixBrownBag.log
-rw-r--r-- 1 nichols staff 29073 Apr 11 11:34 UnixBrownBag.tex
```
- Ex ls -l | awk '{print \$6, \$7}'

```
Apr 11
Apr 11
Apr 11
Apr 11
```

58

## Power Programs: awk details

- -F flag specifies a new record separator  
Ex awk -F: '{print \$5}' /etc/passwd
  - The password file is a colon separated database
  - This will select the 5th record, the real user name
- Password file  

```
aspinall:x:5070:10:Jeff Aspinall:/afs/sph.umich.edu/user/a/aspinall:/usr/local/bin/tcsh
dhunsche:x:14852:10:David Hunsche:/afs/sph.umich.edu/user/d/dhunsche:/bin/csh
nichols:x:137710:10:Thomas E. Nichols:/afs/sph.umich.edu/user/n/nichols:/usr/local/bin/tcsh
wluo:x:45403:10:Wen-Lin Luo:/afs/sph.umich.edu/user/w/wluo:/bin/csh
hayasaka:x:103514:10:Satoru Hayasaka:/afs/sph.umich.edu/user/h/hayasaka:/bin/csh
ghoshd:x:138825:10:Debashis Ghosh:/afs/sph.umich.edu/user/g/ghoshd:/bin/csh
```
- Output  

```
Jeff Aspinall
David Hunsche
Thomas E. Nichols
Wen-Lin Luo
Satoru Hayasaka
Debashis Ghosh
```

59

## Power Programs: awk details

- Special Variable name: \$NF
  - Gives you the last record, regardless of how many records
  - Ex awk -F: '{print \$NF}' /etc/passwd
  - Output  

```
/usr/local/bin/tcsh
/bin/csh
/usr/local/bin/tcsh
/bin/csh
/bin/csh
/bin/csh
```

60

## Power Programs: Quiz

---

- `ls -l | grep 'Jun' | awk 'print $NF'`
- 

- What does this script do?

```
foreach f (thisOne.txt thisOther2.txt thisThing.txt)

    mv $f `echo $f | sed 's/this/that/'`
end
```

---

61

## Power Programs: Quiz (con't)

---

- `cat ListofFileNames | sed 's@/tmp@/my/home/tmp@' > NewList`
- 

- What is the difference between these two?

```
sed s/Normal/Gaussian/ paper.tex > paper2.tex
sed s/Normal/Gaussian/g paper.tex > paper2.tex
```

→ Which one would you want to use?

62

## Shell Scripting: Warnings

---

- Shell scripts are powerful in that they automate tasks

→ You can automatically delete all your files!

→ If using “rm” in a script, test carefully!

– I always test scripts first with `rm's echo'ed`

Ex If my script reads `rm $this $that`

I first test it with `echo rm $this $that`

63

## Shell Scripting: Warnings

---

- *Always* make backups before making a change

→ `cp .cshrc .cshrc_save`

- Modify `.cshrc` *very* carefully!

→ After changing it, use another telnet window to make sure you can still log in.

→ If you can't log in in the new telnet window, use your old one to do `mv .cshrc_save .cshrc`

64

## Shell Scripting: Good habits

---

- Always document

→ Any line beginning with `#` is ignored

→ Write lots of comments

→ Write enough so that you can understand *exactly* what's going on six months from now

65

## Conclusions

---

- A little shell knowledge is a dangerous thing!

→ If you mess up `.cshrc`, you cannot log in

- SPH has great UNIX support

→

<http://www.sph.umich.edu/phisa/tech/docs.html>

66