CS395T: Introduction to Scientific and Technical Computing

Instructors:

Dr. Karl W. Schulz, Research Associate, TACC
Dr. Bill Barth, Research Associate, TACC

Outline

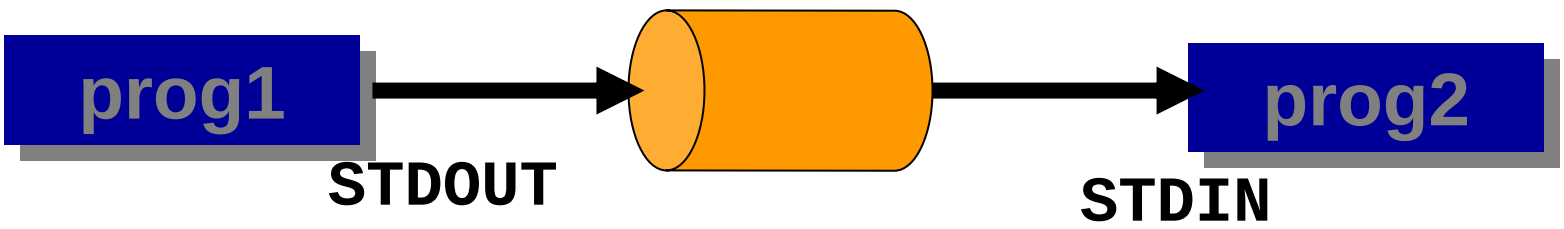
- Administrative Details
  - TA for the class is Vikram Prasannakumar (vikrampkumar@mail.utexas.edu)
  - Office Hours:
    - MW 10:30 – 11:30
    - CS Graduate Lounge in Taylor hall
- Continue with Unix overview
  - Unix pipes
  - Job control
  - Environment Variables
  - Editors
  - Shell Arithmetic
  - Shell scripting
- Assignment #1

Follow-up to Question from Last Week

- **umask** is a built-in shell command used to specify the default permissions mode for newly created files
- It is a three-digit octal mode that represents the permissions that you want to *mask out*
- To determine what permissions a umask will allow, subtract the umask value from the default
- Unix has default permissions for new files and directories:
  - directory is 777 (*remember what that means?)*
  - file is 666
- A common umask value is 022:
  - Default for new directories is thus, 755
  - Default for new files is 644

Unix Pipes

- A pipe is a holder for a stream of data
- A Unix **pipeline** is a set of processes chained by their standard streams, so that the output of each process (*stdout*) feeds directly as input (*stdin*) of the next one
- This is handy for using multiple unix commands together to perform a task



Building Commands

- More complicated commands can be built up by using one or more pipes
- Use the "**|**" character to *pipe* two commands together
- The shell takes care of all the hard work for you
- Example:

```
> cat apple.txt
core
worm seed
jewel

> cat apple.txt | wc
3 4 21
```

Note: the wc command prints the number of newlines, words, and bytes in a file

Job Control

- **The shell allows you to manage *jobs***
  - place jobs in the *background*
  - move a job to the *foreground*
  - suspend a job
  - kill a job
- If you follow a command line with "&", the shell will run the *job* in the background
  - this is you useful if you don't want to wait for the job to complete
  - you can type in a new command right away
  - you can have a bunch of jobs running at once

```
> cat foo | sort | uniq > saved_sort &
```

Listing Your Jobs

- The command *jobs* will list all background jobs:

```
> jobs
[1] Running  cat foo | sort | uniq >
saved_ls &
```

- The shell assigns a number to each job (in this case, the job number is 1)

Managing Jobs

- You can suspend the foreground job by pressing ^Z (Ctrl-Z)
  - Suspend means the job is stopped, but is not dead
  - The job will show up in the **`jobs`** output.

- You can *kill* the foreground job by pressing ^C (Ctrl-C).

- You can also kill a job in the background using the *kill* command (and the appropriate job index)

```
> kill %1
```

Note: it's important to include the "%" sign to reference a job number.

More Job Control Commands

- The **fg** command will move a job to the foreground.
  - You give **fg** a job number (as reported by the **jobs** command)

```
> jobs
[1] Stopped          ls -lR > saved_ls &
> fg %1
ls -lR > saved_ls
```

- What happens if you start a command and then want to place it in the background?
  - Use ^-Z to suspend the command
  - Use the **bg** command to send the job to the background

```
> sleep 60
Suspended
> jobs
[1]  + Suspended                    sleep 60
> bg
[1]    sleep 60 &
> jobs
[1]    Running                      sleep 60
```

Unix Environment Variables

- Unix shells maintain a list of environment variables which have a unique name and a value associated with them
  - some of these parameters determine the behavior of the shell
  - also determine which programs get run when commands are entered (and which libraries they link against)
  - provide information about the execution environment to programs
- We can access these variables:
  - set new values to customize the shell
  - find out the value of some to help accomplish a task

Environment Variables

- To view environment variables, use the env command
- If you know what you are looking for, you can use your new friend grep:

```
> env | grep PWD
PWD=/home/karl
```

- Use the echo command to print variables; the "$" prefix is required to access the value of the variable:

```
> echo $PWD
/tmp
```

- Can also use environment variables in arbitrary commands:

```
Koomie@canyon--> ls $PWD
foo1  foo2
```

Special Environment Variable: PATH

- Each time you provide the shell a command to execute, it does the following:
  - Checks to see if the command is a built-in shell command
  - If it is not a build-in command, the shell tries to find a program whose name matches the desired command
- How does the shell know where to look on the filesystem?
- The **`PATH`** variable tells the shell where to search for programs (non built-in commands)

Special Environment Variable: PATH

- Example PATH Definition:

```
-> echo $PATH
/home/karl/bin/krb5:/opt/intel/compiler70/ia32/bi
n:/home/karl/bin:/usr/local/apps/mpich/icc/bin:/u
sr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr
/X11R6/bin
```

- The **PATH** is a list of directories delimited by colons (":“)
  - It defines a list and *search order*
  - Directories specified earlier in the **PATH** take precedent; once the matching command is found, the search terminates
- You can add more search directories to your PATH by changing the shell startup files
  - **BASH: export PATH="$PATH":/home/karl/bin**
  - **TCSH: set path = (/home/karl/bin $path)**

Other Important Variables

PWD	*current working directory*
MANPATH	*determines where to find man pages*
HOME	*home directory of user*
MAIL	*where your email is stored*
TERM	*what kind of terminal you have*
PRINTER	*specifies the default printer name*
EDITOR	*used by many applications to identify your choice of editors (eg. vi or emacs)*
LD_LIBRARY_PATH	*specifies a search path for dynamic runtime libraries*

Setting Environment Variables

- The syntax for setting Unix environment variables depends on your shell:
  - **BASH**: use the *export* command
    ```
    > export PRINTER=scully
    > echo $PRINTER
    scully
    ```
  - **TCSH**: use the *setenv* command
    ```
    > setenv PRINTER mulder
    > echo $PRINTER
    mulder
    ```

- Note: environment variables that you set interactively are only available in your current shell
  - If you spawn a new shell (or login again), these settings will be lost
  - To make permanent changes, you should alter the login scripts that affect your particular shell (eg. *.login, .profile, .cshrc, etc...)*

Text Editors

Text Editors

- For programming, we need to make use of available Unix text editors
- The two most popular and available editors are *vi* and *emacs*
- You should familiarize yourself with at least one of the two (*and this let's you enter into the editor wars which is a never-ending debate in the programming community)*
- We will have very short introductions to each....

Vi Overview

- Fundamental thing to remember about vi is that it has two different modes of operation:
  - *Insert* Mode
  - *Command* mode
- The *insert* mode puts anything typed on the keyboard into the current file
- The *command* mode allows the entry of commands to manipulate text. These commands are usually one or two characters long, and can be entered with few keystrokes
- Note that vi starts out in the *command* mode by default

Vi Overview

- Quick Start Commands
  - > **`vi`**
  - Press i to enable insert mode
  - Type text (*use arrow keys to move around*)
  - Press Esc to enable command mode
  - Press :w <filename> to save the file
  - Press :q to exit vi

Useful vi commands

- :q! – exit without saving the document. Very handy for beginners
- :wq – save and exit
- / <string> – search within the document for text. n goes to next result
- dd – delete the current line
- yy – copy the current line
- p – paste the last cut/deleted line
- :1 – goto first line in the file
- :$ - goto last line in the file
- $ – end of current line, ^ – beginning of line
- % – show matching brace, bracket, parentheses

Additional vi References

- http://www.eng.hawaii.edu/Tutor/vi.html
- http://staff.washington.edu/rells/R110/
- Vi Commands Reference card: http://tnerual.eriogerg.free.fr/vimqrc.pdf

Emacs Overview

- Programmer friendly modes for common languages (C/C++, Fortran, shell scripts, etc)
- Different from vi in that emacs has only one-main mode
- Lots of commands and extremely customizable (using LISP)
- Includes some very sophisticated features if you take the time to learn them:
  - Compile your executables within emacs
  - Interact with your revision control process (eg. CVS)
  - Control RPM software builds
  - Debug your application using gdb

Emacs Overview

- *> emacs myfile opens myfile for editing*
- *Type whatever text you like (use arrow keys to navigate)*
- *C-x C-s (control + x, control + s) – saves the file*
- C-g exits the current command
- C-x u - Undo
- *C-x C-c exit after saving*

Additional Emacs References

- EMACS Tutorial 1
- EMACS Tutorial 2
- Emacs includes its own on-line tutorial; to run issue the following:
  - > *emacs*
  - Then, enter "C-h t", to invoke the on-line emacs tutorial (*that's a "Control-h", followed by a "t")*

Unix Scripting

- Scripting is “easy” - you just place all the Unix commands in a file as opposed to typing them interactively
- Handy for automating certain tasks:
  - staging your scientific applications
  - performing limited post-processing operations
  - any repetitive operations on files, etc...
- Shells provide basic control syntax for looping, if constructs, etc...

Unix Scripting

- Shell scripts must begin with a specific line to indicate which shell should be used to execute the remaining commands in the file:
  - BASH:
    #!/bin/bash
  - TCSH
    #!/bin/tcsh
- Comment lines can be included if they start with #
- In order to run a shell-script, it must have execute permission. Consider the following script:

```
> cat hello.sh
#!/bin/bash
echo “hello world”

> ./hello.sh
./hello.sh: Permission denied.
```

> chmod 700 hello.sh
> ./hello.sh
hello world

Unix Scripting: Arithmetic Operations

- Simple arithmetic syntax depends on the shell:
  - **TCSH**

```
set i1=10
set j1=3
@ k1 = $i1 + $j1  # Note space between @ and k1
echo "The sum of $i1 and $j1 is $k1”
```

  - BASH

```
i1=2
j1=6
k1=$(($i1*$j1))
echo "The multiple of $i1 and $j1 is $k1”
```

- Note, you can also use the **`expr`** command (for both shells). For example:
  - **TCSH:** `set z=`expr $i1 + $j1``
  - **BASH:** `z=`expr $i1 + $j1``

consult man page on `expr` for more details

Unix Scripting: Conditionals

- **Syntax for conditional expressions depends on your choice of shell:**
- **BASH (general format):**

```
if [ condition_A ]; then
        code to run if condition_A true
elif [ condition_B ]; then
        code to run if condition_A false and
        condition_B true
else
        code to run if both conditions false
fi
```

- **TCSH (general format):**

```
if (condition) then
    commands
else if (other condition) then
    commands
else
    commands
endif
```

Unix Scripting: String Comparisons

- string1 = string2 — Test identity
- string1 !=string2 — Test inequality
- -n string — the length of *string* is nonzero
- -z string — the length of *string* is zero

BASH Example:

```
today="monday"
if [ "$today" = "monday" ] ; then
    echo "today is monday"
fi
```

TCSH Example:

```
set today="friday"
if ( "$today" != "monday” )  then
    echo "today is not monday"
endif
```

BASH Integer Comparisons

- int1 –eq int2 Test identity
- int1 –ne int2 Test inequality
- int1 –lt int2 Less than
- int1 –gt int2 Greater than
- int1 –le int2 Less than or equal
- int1 –ge int2 Greater than or equal

BASH Example:

```
x=13
y=25
if [ $x -lt $y ]; then
  echo “$x is less than $y"
fi
```

TCSH Integer Comparisons

- int1 < int2 Less than
- int1 > int2 Greater than
- int1 <= int2 Less than or equal
- int1 >= int2 Greater than or equal
- int1 == int2 Equal to
- int1 != int2 Not equal to

TCSH Example:

```
set x=13
set y=25
if ( $x < $y ) then
  echo “$x is less than $y"
endif
```

Unix Scripting: Common File Tests

- -d file Test if file is a directory
- -f file Test if file is not a directory
- -s file Test if the file has non zero length
- -r file Test if the file is readable
- -w file Test if the file is writable
- -x file Test if the file is executable
- -o file Test if the file is owned by the user
- -e file Test if the file exists
- -z file Test if the file has zero length

BASH Example:

```
if [ -f foo ]; then
  echo "foo is a file"
fi
```

TCSH Example:

```
if ( -d foo.dir ) then
  echo "foo.dir is a directory"
endif
```

Unix Scripting: For loops

- These are useful when you want to run the same command in sequence with different options
- *sh* example:

```
for VAR in test1 test5 test7b finaltest; do
   runmycode $VAR > $VAR.out
done
```

- *csh* example:

```
foreach VAR ( test1 test5 test7b finaltest )
   runmycode $VAR > $VAR.out
end
```

- *sh* one-liner:

```
for i in `seq 1 5`; do echo $i; done
1
2
3
4
5
```

Quoting in Unix

- We've seen that some metacharacters are treated special on the command line: `* ?`
- What if we don't want the shell to treat these as special - we really mean *, not all the files in the current directory
- To turn off special meaning - surround a string with double quotes:

```
> echo here is a star "*"
here is a star *
```

Use of Quotes

- You have to be careful with the use of different styles of quotes in your commands or scripts

- They have different functions:
  - Double quotes inhibit wildcard replacement only
  - Single quotes inhibit wildcard replacement, variable substitution and command substitution
  - Back quotes cause command substitution

Double Quotes

- Double quotes around a string turn the string in to a *single* command line parameter:

```
> ls
fee file? foo
> ls "foo fee file?"
ls: foo fee file?: No such file or directory
```

- Double quotes only inhibit wildcards; use \ to escape special characters:

```
 > echo “This is a quote \" “
This is a quote “
```

Single Quotes

- Single quotes are similar to double quotes, but they also inhibit variable substitution and command substitution
- Means that special characters do not have to be escaped:

```
> echo 'This is a quote \" '
This is a quote \"
```

Back Quotes

- If you surround a string with back quotes, the string is replaced with the result of running the command in back quotes:

```
> echo `ls`
foo fee file?

> echo "It is now `date` and OU is still
questionable”
It is now Tue Sep 19 11:24:25 CDT 2006 and OU
is still questionable
```

More Quote Examples

- Some Quoting Examples:

```
$ echo Today is date
Today is date
$ echo Today is `date`
Today is Thu Sep 19 12:28:55 EST 2002
$ echo ”Today is `date`”
Today is Thu Sep 19 12:28:55 EST 2002
$ echo ’Today is `date`’
Today is `date`
```

“ “ = double quotes
‘ ‘ = single quotes
` ` = back quotes

Command-Line Parsing

- To build generic shell scripts, consider using command-line arguments to provide the inputs you need internally (syntax again depends on the choice of shell)
- Syntax:
  - $# *refers to the number of command-line arguments*
  - $0 *refers to the name of the calling command*
  - $1, $2, ..., $N *refers to the Nth argument*
  - $* *refers to all command-line parameters*

```
echo "Calling command is:          $0"
echo "Total # of arguments is:     $#"
echo "A list of all arguments is: $*"
echo "The 2nd argument is:         $2"

> ./foo.sh texas rose bowl
Calling command is:          ./foo.sh
Total # of arguments is:     3
A list of all arguments is: texas rose bowl
The 2nd argument is:         rose
```

In tcsh, you can also reference individual arguments with $argv: eg. $1 = $argv[1]

More UNIX Commands for Programmers

– man –k	Search man pages by topic
– time	How long your program took to run
– date	print out current date/time
– test	Compare values, existence of files, etc
– tee	Replicate output to one or more files
– diff	Report differences between two files
– sdiff	Report differences side-by-side
– wc	Show number of lines, words in a file
– sort	Sort a file line by line
– gzip	Compress a file
– gunzip	Uncompress it
– strings	Print out ASCII strings from a (binary)
– ldd	Show shared libraries program is linked to
– nm	Show detailed info about a binary obj
– tar	Archiving utility
– uniq	Remove duplicate lines from a sorted file
– which	Show full path to a command
– file	Determine file type

Assignment

- Assignment #1 has been uploaded to blackboard
  - It is due a week from Friday, September 29, 2006
  - Motivation is to use Lonestar to perform some simple commands and write a shell-script
  - Use the accounts provided
- Important Note:
  - don’t try to login until after this weekend
  - Lonestar is out of production this week
  - We’ll send out an email when you can login again

References/Acknowledgements

- National Research Council Canada (Rob Hutten, Canadian Bioinformatics Resource)
- *Intro. to Unix*, Dave Hollinger, Rensselaer Polytechnic Institute
- Bash Reference Manual, http://www.faqs.org/docs/bashman/bashref.html
- Advanced Bash-Scripting Guide, http://db.ilug-bom.org.in/Documentation/abs-guide/
- TCSH Reference, http://www.tcsh.org/tcsh.html/top.html
- **Unix in a Nutshell**, A. Robbins, O'Reilly Media, 2006.