SHARCNET™

Automating Tasks Using bash

David McCaughan, *HPC Analyst*
SHARCNET, University of Guelph
`dbm@sharcnet.ca`

SHARCNET™

Overview

- Introduction to command shells & bash
- bash fundamentals
  - I/O redirection, pipelining, wildcard expansion, shell variables
- Shell scripting
  - writing bash scripts
  - control structures, string operations, pattern matching, command substitution
  - system tools
- Examples
  - demo

SHARCNET™

What is a Shell?

- User interfaces
  - GUI, character based, etc.
- A *shell* is a character-based user interface
  - interprets the text typed in by the user translating them into instructions to the operating system (and vice versa)
  - anyone using SHARCNET systems is already familiar with the *command line* your shell provides (typically bash)
- We tend to see a shell purely as a *user interface*
  - possible to use it as a programming environment also
  - shell *scripts*

SHARCNET™

Brief History of the Major UNIX Shells

- 1979: **Bourne shell** (`sh`)
  - first UNIX shell
  - still widely used as the LCD of shells
- 1981: **C shell** (`csh`)
  - part of BSD UNIX
  - commands and syntax which resembled C
  - introduced aliases, job control
- 1988: **Bourne again shell** (`bash`)
  - developed as part of GNU project (default shell in Linux)
  - incorporated much from csh, ksh and others
  - introduced command-line editing, functions, integer arithmetic, etc.

SHARCNET™

bash Basics

- Review of concepts
  - bash has a great deal of syntax that you may already be using in your command lines
    - I/O redirection, pipelines, wildcard expansion
  - anything we do on the CLI applies equally to scripts (remember, our command-line is provided by a bash shell!)
- Live review
  - use "help" command to obtain a list of commands, and specific information on any built-in command

SHARCNET™

Reminder: System Tools

- Anything that is usable on the system, can be used in a script---consider some commonly used utilities:
  - ***echo*** (output text to stdout)
    - e.g. `echo "Hello, world!"`
    - e.g. `echo -n "Hello, world!"`
  - ***cat*** (copy input to output)
    - e.g. `cat somefile.txt`
  - ***cut*** (select columns from text)
    - e.g. `cut -f 2 -d ' ' file_notable_field2.txt`
  - ***sed*** (stream editor)
    - e.g. `sed -e 's/\  */\ /g' file_excess_ws.txt`
  - ***mv***, ***cp***, ***mkdir***, ***ls***, ***file***, etc.

SHARCNET™

I/O Redirection

- When we run a program we always have the notion of "standard input" and "standard output"
  - typically the keyboard and terminal respectively
- Redirecting the input/output streams

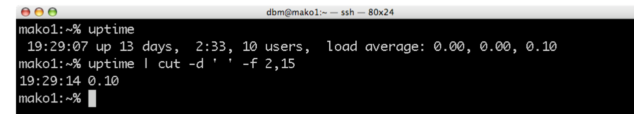
```
./myprog arg1 arg2 > output.txt
./myprog arg1 arg2 < input.txt
./myprog arg1 arg2 < input.txt > output.txt
```

  - see also:
    - `>` vs `>>` (overwrite vs append)
    - `1>` `2>` (stdout [default], stderr)

SHARCNET™

Pipelineing

- System calls exist to allow the programmer to connect *stdout* of one process to *stdin* of another
  - bash provides a means of doing this on the command-line; we refer to this as "piping" the output of the first to the input of the second
  - e.g. grabbing just the time and load average for past 15min from output of `uptime` command, using `cut`:



SHARCNET™

Wildcard Expansion

- bash will expand certain meta-characters when used in file names
  - `?` - matches any single character
  - `*` - matches any sequence of characters, including none
  - `[]` - matches any character in the set (first char `!` negates)
- Note that this expansion is performed by the shell

```
dbm@mako1:~ — ssh — 80x24
mako1:~/pub/exercises/heatflow% ls -l
total 28
-rw-r--r-- 1 dbm dbm 1686 2006-02-13 11:08 common.c
-rw-r--r-- 1 dbm dbm  454 2006-02-13 11:08 common.h
-rw-r--r-- 1 dbm dbm 3845 2006-02-13 11:08 heatflow.c
-rw-r--r-- 1 dbm dbm  864 2006-02-13 11:08 Makefile
-rw-r--r-- 1 dbm dbm 5468 2006-02-13 11:08 PoleSegment.c
-rw-r--r-- 1 dbm dbm 1493 2006-02-13 11:08 PoleSegment.h
mako1:~/pub/exercises/heatflow% ls -l common.?
-rw-r--r-- 1 dbm dbm 1686 2006-02-13 11:08 common.c
-rw-r--r-- 1 dbm dbm  454 2006-02-13 11:08 common.h
mako1:~/pub/exercises/heatflow% ls -l *.c*
-rw-r--r-- 1 dbm dbm 1686 2006-02-13 11:08 common.c
-rw-r--r-- 1 dbm dbm 3845 2006-02-13 11:08 heatflow.c
-rw-r--r-- 1 dbm dbm 5468 2006-02-13 11:08 PoleSegment.c
mako1:~/pub/exercises/heatflow% ls -l *.[ch]
-rw-r--r-- 1 dbm dbm 1686 2006-02-13 11:08 common.c
-rw-r--r-- 1 dbm dbm  454 2006-02-13 11:08 common.h
-rw-r--r-- 1 dbm dbm 3845 2006-02-13 11:08 heatflow.c
-rw-r--r-- 1 dbm dbm 5468 2006-02-13 11:08 PoleSegment.c
-rw-r--r-- 1 dbm dbm 1493 2006-02-13 11:08 PoleSegment.h
mako1:~/pub/exercises/heatflow%
```


SHARCNET™

A Note About Meta-characters

- bash recognizes many characters with “special meaning”
  - already we’ve seen: `> | * ? [ ]`
  - there are many more:
    - `~` - home directory
    - `#` - comment
    - `$` - variable
    - `&` - background job
    - `;` - command separator
    - `’` - strong quotation (no interpretation)
    - `”` - weak quotation (limited interpretation)
    - - whitespace
    - etc.

HPC Resources

SHARCNET™

A Note About Meta-characters (cont.)

- <u>Quotes</u>
  - enclosing a string in single-quotes will prevent the shell from interpreting them

```
mkdir ‘Name With Spaces’
cat ‘filenamewitha*.txt’
```

- <u>Escaping characters</u>
  - a backslash “escapes” meta-character that follows
    - consider: line continuation, literal quotes in strings, etc.

```
cat filenamewitha\*.txt
```

HPC Resources

SHARCNET™

Shell Variables

- A shell variable is a name with an associated string value
  - you have likely already seen these in your shell as *environment variables* (PATH, LD_LIBRARY_PATH, etc.)
  - by convention we use all upper-case for shell variables, however it is common to see lower case “temporary” variables in scripts
- Shell variables are created by assignment

  `VARNAME=string`
  - note: no whitespace around `=` (most common error)
  - a variable can be “deleted”, if necessary, using `unset`
    - unknown variables are assumed to be the empty string

HPC Resources

SHARCNET™

Shell Variables (cont.)

- The value of a shell variable can be used in commands by enclosing the name in `${}`
  - this is very easy to play with on the command-line (and excellent way to distinguish single and double quotes)

```
dbm@mako1:~ — ssh — 80x24
mako1:~% echo ${PI}

mako1:~% PI=3.14
mako1:~% echo ${PI}
3.14
mako1:~% APPROXIMATELY_PI=3
mako1:~% echo "Is ${APPROXIMATELY_PI} really close enough to ${PI}?"
Is 3 really close enough to 3.14?
mako1:~% echo 'Is ${APPROXIMATELY_PI} really close enough to ${PI}?'
Is ${APPROXIMATELY_PI} really close enough to ${PI}?
mako1:~% ls tmp
DDT_OPT
mako1:~% mkdir tmp/newdir_${APPROXIMATELY_PI}
mako1:~% ls tmp
DDT_OPT  newdir_3
mako1:~%
```


SHARCNET™

Shell Programming

- In theory we could write increasingly complex command-lines to produce sophisticated behaviours
  - this would quickly become impractical
  - we will write shell *scripts* to facilitate more complex situations
- **Script**
  - a file containing shell commands
  - created using a text editor
  - can contain any legal bash commands
    - i.e. everything you are already used to being able to do on the command-line, together with the bash shell features you are learning today (and much more)

HPC Resources

SHARCNET™

Running a Script

- Instruct your shell to execute the contents of a text file as bash commands
  `source scriptname`
  - executes lines of file as commands in your current shell (as if you'd typed them in at the command-line)
- More convenient to run them like a program
  `#!/bin/bash`
  - should be first line of script (portability)
  - set execute permission on the file (`chmod u+x scriptname`)
  - run it as if it were any other program
  - note: this executes commands in a *new* shell

HPC Resources

SHARCNET™

Example: Running a Script

```
dbm@mako1:~ — ssh — 80x24
#!/bin/bash

RECIPIENT=world
DIRCMD='ls -l'

echo "Hello, ${RECIPIENT}!"
echo "Here are the contents of the current directory:"
${DIRCMD}
echo
echo "Here are the contents in reverse order:"
${DIRCMD} | sort -r
```

```
dbm@mako1:~ — ssh — 80x24
mako1:~/tmp% ls -l
total 8
drwxr-xr-x 3 dbm dbm 4096 2007-02-27 11:53 DDT_OPT
-rwxr-xr-x 1 dbm dbm  210 2007-04-01 21:28 hello
mako1:~/tmp%
mako1:~/tmp% ./hello
Hello, world!
Here are the contents of the current directory:
total 8
drwxr-xr-x 3 dbm dbm 4096 2007-02-27 11:53 DDT_OPT
-rwxr-xr-x 1 dbm dbm  210 2007-04-01 21:28 hello

Here are the contents in reverse order:
total 8
-rwxr-xr-x 1 dbm dbm  210 2007-04-01 21:28 hello
drwxr-xr-x 3 dbm dbm 4096 2007-02-27 11:53 DDT_OPT
mako1:~/tmp%
```

HPC Resources

Control Structures

- We need a means of performing branching and managing flow of control to be truly useful
- Branching:
  - *IF..ELSE*
  - Also: *CASE*
- Iteration:
  - *FOR*
  - Also: *WHILE, UNTIL, SELECT*

Branching: *IF + conditions*

```
if condition; then
    commands
[elif condition; then
    commands
… ]
[else
    commands]
fi
```

- *Note:*
  `if condition`
  `then`
  - *is equivalent*

- `condition`
  - any list of commands
  - can link conditions using &&, ||
  - ***if*** tests the exit status of the last command;
  - i.e. "*if program execution succeeds then do the following*"
- syntax: `[ condition ]`
  - `[]` is a statement; returns an exit status corresponding to truth of *condition*
  - necessary as ***if*** can only test exit status

Condition Tests

- *String (i.e. variable) testing*
  - e.g. `[ str1 = str2 ]`
    `str1 = str2` - equal
    `str1 != str2` - not equal
    `str1 < str2` - less than
    `str1 > str2` - greater than
  - unary tests for null strings
    `-n str` - not null
    `-z str` - is null
- *File testing*
  - e.g. `[ -e ${filename} ]`
    `-e` - file exists
    `-d` - file exists + is directory
    `-f` - file exists + is regular
    `-r` - have read perm.
    `-w` - have write perm.
    `-x` - have execute perm.
  - binary tests for modification time:
    `[ file1 -nt file2 ]`
    `[ file1 -ot file 2 ]`

IF Examples

```
#
# detect failure in attempt to copy ${infile} to ${outfile}
#
if ( ! cp ${infile} ${outfile} >& /dev/null ); then
        echo "error copying input file to output location"
        exit 2
fi


#
# test if ${dir} a directory; shows a compound condition
#
if [ ${dir} = ${targetdir} && -d ${dir} ]; then
    mv ${dir}/${file} ${archivedir}
elif [ ${dir} = ${targetdir} && -f ${dir} ]; then
    echo "${dir} is a file"
else
    echo "${dir} does not exist"
if
```

SHARCNET™

Iteration: *FOR*

```
for name [in list]; do
    commands-can use $name
done
```

- *Note:*
  `list`
  - a whitespace separated list of words
  - if omitted, `list` defaults to "`$@`", the list of command-line arguments (which we haven't discussed)
- operation
  - names in `list` are iteratively assigned to the variable `name`, and the body of the loop is executed for each
- counting loops cannot be implemented with this type of loop
  - traditionally use while or until loops when counting is necessary
  - far more convenient to be able to iterate over values when processing files, etc.

SHARCNET™

FOR Examples

```
#
# simple example makes a few directories
#
for course in CIS1000 MATH200 CHEM1010; do
    mkdir ${course}
done;


#
# submit program in all directories in current directory to
# queues five times each (note use of list of numbers as names)
#
DIRLIST=$(ls)

for dir in ${DIRLIST}; do
    for trial in 1 2 3 4 5; do
        echo "Submitting trial ${trial} from ${dir}..."
        sqsub -q serial -o ${trial}-OUT.txt ./prog ${trial}-IN.txt
    done;
done;
```

SHARCNET™

String/Pattern Matching

Key to pulling apart pathnames
(long, but good, definitions from O'Reily '98), easy example follows:

Operator	Definition
`${variable#pattern}`	If *pattern* matches **beginning** of *variable*'s value, delete the <u>shortest part</u> that matches and return the rest
`${variable##pattern}`	If *pattern* matches **beginning** of *variable*'s value, delete the <u>longest part</u> that matches and return the rest
`${variable%pattern}`	If *pattern* matches **end** of *variable*'s value, delete the <u>shortest part</u> that matches and return the rest
`${variable%%pattern}`	If *pattern* matches **end** of *variable*'s value, delete the <u>longest part</u> that matches and return the rest

HPC Resources

SHARCNET™

String/Pattern Matching Examples

dbm@mako1:~ — ssh — 80x24

```
#!/bin/bash

BEG_LONG=${FULLPATH##/*/}
BEG_SHORT=${FULLPATH#/*/}
END_SHORT=${FULLPATH%.*}
END_LONG=${FULLPATH%%.*}

echo "## long =                   $BEG_LONG"
echo "# short =       $BEG_SHORT"
echo "full    = $FULLPATH"
echo "% short = $END_SHORT"
echo "%% long = $END_LONG"
```

dbm@mako1:~ — ssh — 80x24

```
mako1:~/usr/sharcnet% ./patheg
## long =                   benchmark.hpl.20070329
# short =       dbm/sharcnet/benchmark.hpl.20070329
full    = /home/dbm/sharcnet/benchmark.hpl.20070329
% short = /home/dbm/sharcnet/benchmark.hpl
%% long = /home/dbm/sharcnet/benchmark
mako1:~/usr/sharcnet%
```

HPC Resources

Command Substitution

- A means of representing the output from other programs into a shell variable
- **$(command)**
  - executes the command in brackets
  - expression is replaced with stdout from the command
  - compare with the archaic ` (as a pre-execute)
- e.g.

```
CURDIR=$(pwd)
FILETYPE=$(file ${filename})
for file in $(ls); do …
```

Some Examples

Demo

Example: Job Submission

- Monte Carlo-type simulations
  - once the experiment is designed and parameters set, we need to submit vast numbers of jobs to the queue
  - can speed this process dramatically using a script to do the submissions
  - Notes:
    - this is most easily accomplished by having the program take its parameters either on the command-line, or from a file that is specified on the command-line; similarly, output should either go to stdout or to a file specified on the command-line
      - makes it easy to submit from the same directory
  - "for each set of parameters, submit a job with those parameters"

Example: Job Submission (simple, parameter-based)

```
#!/bin/bash

#
# DEST_DIR is the base directory for submission
# EXENAME is the name of the executable
#
DEST_DIR=/work/dbm/MC2
EXENAME=hello_param

cd ${DEST_DIR}
for trial in 1 2 3 4 5; do
    for param in 1 2 3; do
        echo "Submitting trial_${trial} - param_${param}..."
        sqsub -q serial -o OUTPUT-${trial}.${param}.txt \
                               ./${EXENAME} ${trial}-${param}
    done;
done;
```

SHARCNET™

Example: File Management

- Monte Carlo-type simulation with hard coded parameters/files
  - this is essentially the same problem, with the added problem of potentially needing a separate directory/executable for every submission
    - do we have the option to recode the application to better handle its parameters?
- This was a real issue for a user: what we ended up with, was a basic set of directories
  - each contained the relevant executable/input file for a given test
  - we needed N replications of each, and due to a hard coded output file it had to run from its own directory for max. potential parallel execution
  - script 1: *copy/propagate the basic set of directories to N replications*
  - script 2: *submit all jobs from the appropriate directories*

SHARCNET™

Example: File Management (submission set-up)

```
#!/bin/bash
#
# SRC_DIR is the location of the directories containing runs
# DEST_DIR is the location for expanded set-up (run1-10)
# ***************************************************************
# *** SRC_DIR should never be the same as DEST_DIR unless you ***
# *** don't like your files, or other users of the system     ***
# ***************************************************************
#
SRC_DIR="/work/dbm/MC2src"
DEST_DIR="/work/dbm/MC2"

for runtype in $(ls ${SRC_DIR}); do
    for run in 1 2 3 4 5 6 7 8 9 10; do
        echo "Processing ${runtype} - run${run}..."
        mkdir -p ${DEST_DIR}/${runtype}/run${run}
        cp -R ${SRC_DIR}/${runtype}/* \
                     ${DEST_DIR}/${runtype}/run${run}
    done;
done;
```

SHARCNET™

Example: Job Submission (explicit, by directory)

```
#!/bin/bash

#
# DEST_DIR is the base directory for expansion
# EXENAME is the name of the executable
#
DEST_DIR=/work/dbm/MC1
EXENAME=hello

for dir in $(ls ${DEST_DIR}); do
    for subdir in $(ls ${DEST_DIR}/${dir}); do
        echo "Submitting ${dir} - ${subdir}..."
        cd ${DEST_DIR}/${dir}/${subdir}"
        sqsub -q serial -o OUTPUT.txt ./${EXENAME}"
    done;
done;
```

SHARCNET™

Common Got'chas

- There is no whitespace around = in variable assignment
  - correct: `VAR=value` error: `VAR = value`
- There is whitespace between conditional brackets and their content
  - correct: `if [ -d ${dirname} ]; then`
  - error: `if [-d ${dirname}]; then`
- Although you often get away without curly braces around variable names, it is a bad habit that will eventually break on you
  - correct: `${varname}` avoid: `$varname`
- Failing to "test drive" your script-constructed commands using `echo` is asking for trouble

SHARCNET™

Exercise: Text Processing

The purpose of this exercise is to allow you to practice writing a simple bash shell script, incorporating appropriate system tools to perform text processing on data files.

SHARCNET™

Exercise

1) Reformatting text is a pervasive issue for computational techniques. A number of data files are in `~dbm/pub/exercises/bash`. These might be end-of-run results from a number of simulation runs. We are only interested in the "Global Error" value, and the parameters used for the run in question, for the next step in our analysis. Note that the file names explicitly encode the parameters for the run (N, G, X)
   - write a bash script to extract the "Global Error" value from all data files, summarizing them in a single file, one per line, together with the parameters used for the run.
   - a line in the post-processed file should look as follows (where # is the value of the execution parameter encoded in the file name):
     - `N:#   G:#   X:#   Error = #`
2) Consider:
   - pattern/string matching operations for extracting parameters from file names
   - recall ">>" redirects output to a file, *appending* it if the file already exists

SHARCNET™

Exercise (cont.)

- Answer the following questions:
  - *what system utilities did you leverage to accomplish this task? Were there alternatives?*
  - *what changes would be required if the data files being processed were spread through a directory hierarchy (rather than all in one directory)? what if the parameters were contained within the file rather than as part of the file name?*
  - *For a "take home exercise", change your script so that it accepts the list of file names to be processed on the command-line (you will need to look up how to handle command-line parameters in a bash shell script).*

SHARCNET™

A Final Note

- We have deliberately omitted vast detail regarding bash
  - customizing the interactive shell environment
  - command-line options, functions, parameters, etc.
  - we focused on only common SHARCNET user tasks
- For additional information:
  - "`help`" command in a bash shell
  - bash man page
  - GNU bash documentation
    - `http://www.gnu.org/software/bash/manual/bash.html`
  - "*Learning the bash Shell (2e)*", C. Newham and Bill Rosenblatt, O'Reilly & Associates, 1998.