Introduction to Linux Scripting (Part 2)

Brett Milash and Wim Cardoen

CHPC User Services

Overview

- Advanced Scripting
- Compiling Code

Getting the exercise files

- For today's exercises, open a session to linuxclass.chpc.utah.edu or one of the cluster head nodes and run the following commands:

```
cp ~u0424091/LinuxScripting2.tar.gz .
tar  xvfz  LinuxScripting2.tar.gz
cd LinuxScripting2/
```

Capturing output of a command

- The output of a command can be put directly into a variable with the backtick: `
- The backtick is not the same as a single quote:

- Bash form: `VAR=`wc -l $FILENAME``
- Tcsh form: `set VAR="`wc -l $FILENAME`"`

Dates and Times

- Date strings are easy to generate in Linux
  - “date” command gives the date, but not nicely formatted for filenames
  - “date --help” or “man date” will show format options
- A nice formatted string format (ns resolution):

```
$ date "+%Y-%m-%d_%k-%M-%S_%N"
2018-09-18_10-08-57_791967522
```

- You can put other things in the format string:

```
$ date "+Today is %A %m-%d-%Y”
Today is Tuesday 09-18-2018
```

Exercise 2.1

Modify your previous script so that instead of writing to an output file with a fixed name, the output filename is derived from the input file (e.g., ‘filename” becomes “filename.date”). Don’t forget to copy your script in case you make a mistake!

Command execution to string - **VAR=`command`** (use the backtick)

Dates - **date “+%Y-%m-%d_%k-%M-%S_%N”** (or pick your own format)

Solution to Exercise 2.1

```
#!/bin/bash
INPUT=$1
DATE=`date +%Y-%m-%d_%k-%M-%S_%N`
OUT=”$INPUT.$DATE”
grep ‘\!’ $INPUT > $OUT
wc –l $OUT
```

```
#!/bin/tcsh
set INPUT = $1
set DATE = "`date +%Y-%m-%d_%k-%M-%S_%N`"
set OUT = “$INPUT.$DATE”
grep ‘\!’ $INPUT > $OUT
wc -l $OUT
```

Every time you run the script, a new unique output file should have been generated.

If statements

```
#!/bin/bash
VAR1="name"
VAR2="notname”
dirname=“.”
if [ $VAR1 == $VAR2 ]
then
  echo ”VAR1 and VAR2 are equal"
else
  echo ”VAR1 and VAR2 not equal"
fi
if [ -d $dirname ]
then
  echo ”$dirname is a directory!”
fi
```

```
#!/bin/tcsh
set VAR1 = "name"
set VAR2 = "notname”
set dirname = “.”
if ($VAR1 == $VAR2) then
  echo ”VAR1 and VAR2 are equal"
else
  echo ”VAR1 and VAR2 not qual"
endif
if ( -d $dirname ) then
  echo ”$dirname is a directory!"
endif
```

- The operators ==, !=, &&, ||, <, > and a few others work.
- You can use if statements to test two strings, or test file properties.

Testable file properties

Test	bash	tcsh
Is a directory	`-d`	`-d`
If file exists	`-a,-e`	`-e`
Is a regular file (like .txt)	`-f`	`-f`
Readable	`-r`	`-r`
Writeable	`-w`	`-w`
Executable	`-x`	`-x`
Is owned by user	`-O`	`-o`
Is owned by group	`-G`	`-g`
Is a symbolic link	`-h, -L`	`-l`
If the string given is zero length	`-z`	`-z`
If the string is length is non-zero	`-n`	`-s`

-The last two flags are useful for determining if an environment variable exists.
-The rwx flags only apply to the user who is running the test.

Loops (for/foreach statements)

```
#!/bin/bash
for i in 1 2 3 4 5
do
  echo $i
done

for filename in *.c
do
  grep "string" $filename >> out
done
```

```
#!/bin/tcsh
foreach i (1 2 3 4 5)
  echo $i
end
foreach i ( *.in )
  touch "$i:gas/.in/.out/"
end
foreach i ( `cat files` )
  grep "string" $i >> list
end
```

- Loops can be executed in a script --or-- on the command line.
- All loops respond to the wildcard operators *,?,[a-z], and {1,2}
- The output of a command can be used as a for loop input.

find command

- ”find” searches a directory for files matching any number of conditions:
  - File type
  - File permissions
  - File name
  - And many, many others
- Find can execute a command on each found file, or simply print it name

```
# search current dir (and subdirs) for PDF files.
find . –name “*.pdf” -readable –print –exec mv {} my_pdfs \;
```

- -exec command must end with “\;”
- {} represents any file that was found

xargs command

- Xargs reads values from the standard input, then executes some command on each of those values

  find . –name “*.data” –print | xargs gzip

- Commands can be run in parallel – very useful on multi-core machines!

  # Find .dat files, run analyses 5 at a time:

  find . –name “*.data” –print | xargs –P 5 run_analysis

Exercise 2.2

Congratulations – you've inherited files from a very messy project. You need to write a script to clean it up.

1. In LinuxScripting2/ex2.2, create a script to organize the files in the MessyProject directory.
2. Create directories "executables", "source_code", documentation", and "data"
3. Remove any object files (files whose names end with .o)
4. Move any source code files (.c or .h files) to source_code
5. Move any PDF files into "documentation"
6. Move any regular files (not directories) that are executable into "executables"
7. Move any .txt files into "data"

If you need to "reset" your directory, you can extract the files from messyproject.tar:

```
tar xvf messyproject.tar
```

Basic Arithmetic

```
#!/bin/bash
#initialization
i=1
#increment
i=$(( i++ ))
#addition, subtraction
i=$(( i + 2 - 1 ))
#multiplication, division
i=$(( i * 10 / 3 ))
#modulus
i=$(( i % 10 ))
#not math, echo returns "i+1"
i=i+1
```

```
#!/bin/tcsh
#initialization
@ i = 1
#increment
@ i++
#addition, subtraction
@ i = i + 2 - 1
#multiplication, division
@ i = i * 10 / 3
#modulus
@ i = i % 10
#not math, echo returns "i+1"
set i="i+1"
```

- Bash uses **$(( ))**, whereas tcsh uses **@**
- Important! This only works for integer math. If you need more, use python.

Bash “Strict” Mode

- Some bash settings simplify debugging:

```
set –e            # Exit immediately on any error
set –u            # Error if referencing undefined variable
set –o pipefail   # Error on any pipe command

# Example: this code should fail:
pattern=“somestring $some_undefined_variable”
grep $pattern non_existent_file | wc -l
```

- You can do this all at once:

```
set –euo pipefail
```

- See Aaron Maxwell’s blog:
  - http://redsymbol.net/articles/unofficial-bash-strict-mode/
- Also helpful is “set –x”: prints each line before execution

Interpreted vs. Compiled code

- Source code := collection of ***human-readable*** computer instructions written in a programming language
  (e.g. C, C++, Fortran, Python, R, Java,…)
- Executable := ***binary*** program that can be directly executed on a computer
- **Interpreted** languages: the interpreter parses the source code & executes it immediately
- **Compiled** languages: the source code needs to be transformed into an executable through a chain of compilation & linking
- A few examples of both approaches:
  a. interpreted languages: Python, R, Julia, Bash, Tcsh,…
  b. compiled languages: C, C++, Fortran, …

Creating an executable (Low level)

- For compiled languages, the creation of an executable goes through the following steps:
  - *Preprocessing*: the pre-processor takes the source code (.c,.cc,.f90) and “deals” with special statements e.g. #define, #ifdef, #include (C/C++ case)
  - *Compilation*: takes the pre-processor output and transforms it into assembly language (*.s)
  - *Assembly*: converts the assembly code (*.s) into machine code/object code (*.o)
  - *Linking*: the linker takes the object files (*.o) and transforms them into a library (*.a, *.so) or an executable

- Example : simple.c (C source file)
- Pre-processing:
  - cpp simple.c –o simple.i **or**
  - gcc –E simple.c –o simple.i
- Compilation:
  - gcc –S simple.i [–o simple.s]
    # can also use gcc –S simple.c [-o simple.s]
- Assembly phase: creation of the machine code
  - as simple.s –o simple.o **or**
  - gcc –c simple.c [–o simple.o]
    # can also use gcc –c simple.s [-o simple.o]
- Linking: creation of the executable
  - gcc simple.c [-o simple] **or**
    # use ld (the linker as such) -> complicated expression

Regular way (cont.)

- Either in 1 step:
  a. gcc –o simple simple.c
- Or in 2 steps:
  a. gcc –c simple.c
  b. gcc –o simple simple.o

or more **generally (C, C++, Fortan)**:

- 1-step:
  a. $COMPILER –o $EXE $SOURCE_FILES
  (.f90,.c,.cpp)
- 2-step:
  a. $COMPILER –c $SOURCE_FILES
  b. $COMPILER –o $EXE $OBJECT_FILES

Compilers

- Compilers are system-specific, but, there are quite a few vendors (CHPC has all three):
- GNU: `gcc`, `g++`, `gfortran` – open source, free
- Intel: `icc`, `icpc`, `ifort` – commercial but free for academia
- PGI: `pgcc`, `pgCC`, `pgf90` – commercial

Optimization and debugging

- The compiler can perform optimizations that improve performance.
  - common flags `-O3` (GNU), `-fast` (Intel), `-fastsse` (PGI)
  - Beware! -O3,etc can sometimes cause problems (solutions do not calculate properly)
- In order to debug program in debugger, symbolic information must be included
  - flag `-g`
  - The easiest debugging is to just add `printf` or `write` statements (like using echo)

Exercise 2.3

Go to the subdirectory "ex3". There are a few source files in this directory. Compile these programs using the following steps:

1. Compile cpi_ser.c using gcc. Perform the compilation first in **2** steps i.e.
   create first an object file & then an executable.
   Perform the same compilation in **1** step.
2. Try the same for pi3_ser.f. Does it work?
3. Create the object file of ctimer.c with gcc. Then link both object file ctimer.o and pi3_ser.o into an executable using gfortran.
4. Try compiling cpi_ser.c with the optimization flag: –O3
   Compare the timings with the result obtained under 1.

1-step: Compilation + linking:

gcc hello.c -o hello.x *(C source code)*
gfortran hello.f -f hello.x *(Fortran source code)*

2-step process:

Object compilation: **gcc -c hello.c** *(Creates hello.o)*
Linking: **gcc hello.o -o hello.x** *(Links hello.o with sys. libraries into an executable)*

Using optimization: **gcc -O3 hello.c -o helloFast.x**

Solutions to Exercise 2.3

1. Compiling a C program:

   1-step:

   gcc cpi_ser.c -o cpi_ser.x (Time: ~1.625 s)

   2-step:

   gcc –c cpi_ser.c

   gcc –o cpi_ser.x cpi_ser.o

2. Compiling a Fortran program:

   2-step:

   gfortran –c pi3_ser.f

   gfortran –o pi3_ser.x pi3_ser.o -- Errors (Missing dependencies)

3. Compiling the missing dependency + linking:

   gcc –c timer.c # (creates ctimer.o)

   gfortran ctimer.o pi3_ser.o –o pi3_ser.x

4. Compiling with –O3:

   gcc –O3 cpi_ser.c –o cpi_ser.fast.x

   or:

   gcc –c –O3 cpi_ser.c

   gcc –o cpi_ser.fast.x cpi_ser.o

Next time: compiling serious packages

- Some packages are far more complicated than one or two source files.
  - Many packages use gnu config/make
  - Others use cmake (useful for cross-platform)
  - Others of less repute
- You will almost certainly encounter a package like this if you continue in scientific computing
  - CHPC can help compile programs (can be hard) but knowing how to do it yourself is useful.

Questions?

Email helpdesk@chpc.utah.edu
brett.milash@Utah.edu
wim.cardoen@utah.edu