Introduction to Linux Scripting (Part 2)

Wim Cardoen, David Heidorn and Anita Orendt
CHPC User Services

Overview

- Advanced Scripting
- Compiling Code

Getting the exercise files

- For today's exercises, open a session to one of the cluster interactives and run the following commands:

```
cp ~u0253283/Talks/LinuxScripting2.tar.gz .
tar  -zxvf  LinuxScripting2.tar.gz
cd LinuxScripting2/
```

Commands to string

- The output of a string 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`"`

String replacement

A neat trick for changing the name of your output file is to use string replacement to mangle the filename.

`#!/bin/bash`
`IN="myfile.in"`
`#changes myfile.in to myfile.out`
`OUT=${IN/.in/.out}`
`./program < $IN > $OUT`
	`#!/bin/tcsh`
	`set IN = "myfile.in"`
	`#changes myfile.in to myfile.out`
	`set OUT="$IN:gas/.in/.out/"`
	`./program < $IN > $OUT`

- In tcsh the 'gas' in "`$VAR:gas/search/replace/`" means to search and replace <u>all</u> instances ("global all substrings"); there are other options (use "man tcsh").
- In bash, `${VAR/search/replace}` is all that is needed.
- You can use 'sed' or 'awk' for more powerful manipulations.

Dates and Times

- Date strings are easy to generate in Linux
  - "date" command gives the date, but not nicely formatted for filenames
  - date --help will give format options (use +)
- A nice formatted string format (ns resolution):

```
date +%Y-%m-%d_%k-%M-%S_%N
"2014-09-15_17-27-32_864468693"
```

- For a really unique string, you can use the following command to get a more or less unique string (not recommended for cryptographic purposes)

```
$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' | fold -w 32 | head -n 1)
```

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., ‘file.out” becomes “file.date”). Don’t forget to copy your script in case you make a mistake!

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

Bash replacement – **${VAR/search/replace}**

Tcsh replacement – **“$VAR:gas/search/replace/”**

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/out/}$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:gas/out//”$DATE`
	`grep ‘\!’ $INPUT > $OUT`
	`wc -l $OUT`

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

Conditionals (If statements)

```
#!/bin/bash
VAR1="name"
VAR2="notname"
if [[ $VAR1 == $VAR2 ]]; then
  echo "True"
else
  echo "False"
fi
if [[ -d $VAR ]]; then
  echo "Directory!
fi
```

```
#!/bin/tcsh
set VAR1="name"
set VAR2="notname"
if ($VAR1 == $VAR2) then
  echo "True"
else
  echo "False"
endif
if ( -d $VAR ) then
  echo "Directory!"
endif
```

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

Conditionals (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 i in *.in; do
  touch ${i/.in/.out}
done
for i in `cat files`; do
  grep "string" $i >> list
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.

Exercise 2.2

Run the script called ex2.sh. This will generate a directory "ex2" with 100 directories and folders with different permissions. Write a script that examines all the directories and files in "ex2" using conditionals and for loops. For each iteration of the loop:

1. Test if the item is a directory. If it is, delete it.
2. If the file is not a directory, check to see if it is executable.
   A. If it is, then change the permissions so the file is not executable.
   B. If the file is not executable, change it so that it is executable and rename it so that it has a ".script" extension.
3. After all the files have been modified, execute all the scripts in the directory.

For loops - Bash : **for VAR in *; do ... done**
Tcsh : **foreach VAR (*) ... end**

If statements - Bash : **if [[condition]]; then ... elif ... else ... fi**
Tcsh : **if (condition) then ... else ... else if ... endif**

Useful property flags - **-x** for executable, **-d** for directory

-You can reset the directory by re-running the script ex2.sh

-Make sure that you do not write your script in the ex2 directory, or it will be deleted!

Solution to Exercise 2.2

```
#!/bin/bash
for i in ex2/*; do
  if [[ -d $i ]]; then
    rm -rf $i
  else
    if [[ -x $i ]]; then
      chmod u-x $i
    else
      chmod u+x $i
      mv $i $i.script
    fi
  fi
done
for i in ex2/*.script; do
  ./$i
done
```

```
#!/bin/tcsh
foreach i ( ex2/* )
  if ( -d $i ) then
    rm -rf $i
  else
    if ( -x $i ) then
      chmod u-x $i
    else
      chmod u+x $i
      mv $i $i.script
    endif
  endif
end
foreach i ( ex2/*.script )
  ./$i
end
```

Basic Arithmetic

```bash
#!/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
```

```tcsh
#!/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.

Interpreted vs. Compiled code

- Source := 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.{f90,c,cpp}
  b. $COMPILER –o $EXE $SOURCE.o

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

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.

GNU config and make

- Configure: A scripting utility that checks for certain libraries and applications, as well as compiler capablities, and building makefiles.
  - Executed by the **./configure** script in the package directory.
  - You can use **./configure --prefix=<PATH>** to decide where to install the package, otherwise it will install in the same location as the package source.
- Make: Takes instructions from a makefile (a special script) to compile source in order to make a program.
  - As simple as executing **make** in a folder with a Makefile (or specifying the makefile with **-f**)
  - Sometime you need to use **make install** to finish the compilation process.

Exercise 2.4

You will download and compile the zlib library in this exercise. zlib is used by many programs for file compression.

1. Make a directory called “ex4" and cd to it.
2. Download and untar the zlib library with the following :
   - **wget http://zlib.net/zlib-1.2.8.tar.gz**
   - **tar -zxvf zlib-1.2.8.tar.gz**
3. Enter the newly created dir. + configure zlib so that it installs in the dir. $HOME/myzlib and not the source directory (zlib-1.2.8).
   - **./configure --prefix=$HOME/myzlib**
   - Compile using **make** and then **make install**.
4. Check to see if the library was installed properly in $HOME/myzlib/lib (the files libz.so, libz.a should exist).

Solutions for Exercise 2.4

1. mkdir ex4
2. cd ex4
3. wget http://zlib.net/zlib-1.2.8.tar.gz *# Retrieve the src. code*
4. tar –zxvf zlib-1.2.8.tar.gz *# Unzip + extract the src. code*
5. cd zlib-1.2.8 *# Enter the newly created dir.*
6. ./configure –prefix=$HOME/myzlib
7. make *# Compile + link the code*
8. make install *# If step 7. was OK => install the library*
9. ls –la $HOME/myzlib/lib *# You should see libz.a & libz.so*

Questions?

Email issues@chpc.utah.edu