The bash Shell

ME 4953/5013 - Introduction to High-Performance Computing

Shells Offerings

- Two main categories
  - The Bourne family
    - Bourne (`/bin/sh`), Korn (`/bin/ksh`), Bash (`/bin/bash`)
  - The C Shell (`/bin/csh`)
    - Tsch (`/bin/tcsh`)
- Bash and C are the most common
  - Bash is default on Linux
  - To list your shell invoke `echo $SHELL`

bash pattern matching

- Slightly different than regular expressions used in `grep`

Wild Card	Matches
`*`	Any number of characters including none
`?`	A single character
`[ijk]`	A single character – either `i`,`j`, or `k`
`[x-z]`	A range of characters `x` to `z`
`[!ijk]`	A single character not `i`, `j`, or `k`
`{pat1,pat2,...}`	`pat1`, `pat2`, etc.
`!(flname)`	All except `flname`
`!(flname1|flname2)`	All except `flname1` and `flname2`

Pattern matching examples

Command	Significance
`ls *.lst`	Lists all files with extension .lst
`mv * ../bin`	Moves all files to `bin` subdirectory of parent directory
`gzip .?*.?*`	Compresses all files beginning with a dot, followed by one or more characters, then a second dot followed by one or more characters.
cp chap chap*	Copies file `chap*` (`*` loses meaning here)
`cp ??????  progs`	Copies to `progs` all six-character filenames
`rm note[0-1][0-9]`	Removes files `note00, note01, ...` through `note19`
`ls *.[!o]`	Lists all files having extensions except C object files
`cp ?*.*[!1238]`	Copies to the parent directory files having extensions with at least one character before the dot, but not having `1, 2, 3,` or `8` as the last character.

Escaping and quoting

- When the \ precedes a metacharacter (wildcard) its special meaning is turned off.
  - This is known as *escaping*
- Quoting the metacharacter or even the whole pattern has the same effect of turning off the special meanings.
  - e.g., `rm ''chap*''`
  - e.g., `rm ''My Document.doc''`
    - Contains the space between My and Documents

Command substitution

- The shell supports, in addition to pipes (`|`), another way to join two commands together.
  - Surround the substituted command with single backquotes (`` `pwd` ``)

Example

```
> echo The date today is `date`
```

Shell variables

- A variable assignment is of the form *variable=value* and its evaluation requires the $ prefix

Example

```
> count=5
> echo $count
```

- A variable can be assigned the value of another variable:

Example

```
> total=$count
> echo $total
```

- No special steps are needed to concatenate variables.

Example

```
> ext=.avi
> moviename=holmes
> filename=$moviename$ext
```

Shell scripts

- We store a group of commands in a file and execute them sequentially. These files are called *shell scripts*
- Use `vi` or `emacs` to create the following script:

script.sh

```
directory=‘pwd‘
echo The date today is ‘date‘
echo The current directory is $directory
```

- The extension `.sh` is used by convention.
- Must change the permissions of a shell script to be executed.

The *She-Bang* Line (`#!`)

- The first line of a shell script should contain the full path to the shell you wish to execute, e.g., `#!/bin/bash` or `#!/bin/csh`, etc.
- The login shell reads this line and spawns a sub-shell of the type specified.
  - Can spawn C shells from Bash and vice-versa.

Making scripts interactive

read

- `read` causes the script to pause and accept input from `stdin`

Example

```
1 #!/bin/bash
2 echo -n "Enter the directory to be searched: "
3 read dname
4 echo -n "Enter the file extension to find: "
5 read flext
6 echo Searching for files with extension .$flext in \
7     directory $dname
8 find $dname \( -name "*.$flext" -a -type f \) \
9     2>/dev/null
```

Command line arguments

- Scripts not using `read` can run noninteractively and be used with redirection and pipes.
- *Positional parameters* or command line arguments are useful here.
- The first arguments is stored in `$1`, the second in `$2`, etc.
  - `$#` specifies the number of arguments at command line

Example

```
1 #!/bin/bash
2 echo Searching for files with extension .$2 in \
3     directory $1
4 find $1 \( -name "*.$2" -a -type f \) \
5     2>/dev/null
```

Conditional execution

- `bash` offers conditional blocks, `if`, `else`, `elif` and logical operators *and* (`&&`) and *or* (`||`).

Example

```
1  #!/bin/bash
2
3  if [ $# -eq 1 ] ; then
4      echo Searching for files with extension .$1 in \
5          directory $PWD
6      find $PWD \( -name "*.$1" -a -type f \)  2>/dev/null
7
8  elif [ $# = 2 ] ; then
9      echo Searching for files with extension .$2 in \
10         directory $1
11     find $1 \( -name "*.$2" -a -type f \) 2>/dev/null
12
13 else
14     echo Please specify either 1 or 2 inputs.
15 fi
```

Comparison operators

Numerical comparisons

Operator	Meaning
`-eq (=)`	Equal to
`-ne (!=)`	Not equal to
`-gt (>)`	Grater than
`-ge (>=)`	Greater than or equal to
`-lt (<)`	Less than
`-le (<=)`	Less than or equal to

String testing

Test	If true
`s1 = s2`	String `s1s` equal to `s2`
`s1 != s2`	String `s1` is not equal to `s2`
`-n stg`	String `stg` is not a null string
`-z stg`	String `stg` is a null string
`stg`	`stg` is assigned and not null

File attribute testing

True	True if file
`-f file`	`file` exists and is a regular file
`-r file`	`file` exists and is readable
`-w file`	`file` exists and is writable
`-x file`	`file` exists and is executable
`-d file`	`file` exists and is a directory
`-s file`	`file` exists and has a size greater than zero
`-e file`	`file` exists
`f1 -nt f2`	`f1` is newer than `f2`
`f1 -ot f2`	`f1` is older than `f2`

Final notes on `bash` programming

- `bash` offers `for` and `while` constructs
- Offers integer and floating point computation with `expr` and `bc`, respectively
- These things are *usually* better left to a real programming language, i.e. Python