CS 114 - Fall 2004

Lecture 8 - Friday, October 15, 2004

Shell scripts

Last time, we saw that a file can be made into a program by first making it executable (`chmod +x`) and then prepending a `#!` line:

```
#!command
```

The `#!` is called a *hash bang* or *sh-bang*. When an executable file begins with `#!`, the operating system reads the rest of the line after the `#!` (that is, *command*) and invokes that command, passing the file as an additional argument.

```
% file
```

The operating system will invoke the command:

```
command file
```

Note that the full path to *command* (e.g., `/usr/bin/sed`, not just `sed`) must be given on the `#!` line because the operating system, not the shell interprets the command. The `PATH` is not searched and wildcards are not expanded.

Now, what's a shell script? A shell script is simply a file containing a sequence of shell commands. Any command you can type on the command line—invoking programs, setting variables, etc.,—can be done in a shell script, and vice versa. Making the script file executable and prepending:

```
#!/bin/sh
```

allows the script to be run from the command line. Commands in the file are interpreted using the Bourne shell `/bin/sh`.

Note: you can write shell scripts for shells other than the Bourne shell, but since `/bin/sh` exists on every Unix system, and other shells aren't necessarily present, we will focus on Bourne shell scripts.

A small example

The following script `test.sh` searches the file `file1` for the string "`hi`" and the file `file2` for the string "`hello`", then prints the matching lines in sorted order.

```
$ cat test.sh
#!/bin/sh
fgrep 'hi' file1 > tmp1
fgrep 'hello' file2 > tmp2
cat tmp1 tmp2 | sort
rm tmp[12]
```

Note that shell scripts can do I/O redirection, build pipelines and use wildcards, just like on the command line.

Shell variables

When a shell script is run, it runs in a *different* process than the login shell, even if the login shell is

`/bin/sh`. The shell script *inherits* environment variables from the program that invoked the script (usually the login shell). But, since the script is running in a different process, changes made to the environment in the script are *not* visible to the program that invoked the shell. However, environment variables set in the script *are* inherited by programs invoked by the shell script.

A shell can also have *local variables*. Local variables can be defined in the login shell, or in a shell script. These variables are not inherited by programs the shell invokes.

In the Bourne shell, a local variable is set in exactly the same way as an environment variable. For example, to set the variable `index` to `1`, the following command is used:

```
$ index=1
```

Note there are no spaces around the `=` sign. A local variable is accessed just like an environment variable, as well:

```
$ echo $index
1
```

In fact, the only difference between a local variable and an environment variable is that environment variables are *exported* and local variables are not. The `index` variable can be turned into an environment variable using the `export` command:

```
$ export index
```

Any processes that are invoked later by the shell will inherit the `index` variable. To illustrate the difference:

```
$ envvar=1
$ export envvar
$ echo $envvar
1
$ locvar=2
$ echo $locvar
2
$ sh              # start a new shell
$ echo $envvar
2
$ echo $locvar    # locvar not set, so just echoes a blank line

$ envvar=3
$ echo $envvar
3
$ locvar=4
$ echo $locvar
4
$ exit            # return to the original shell
$ echo $envvar    # echoes the original value
1
$ echo $locvar    # echoes the original value
2
$
```

Substitution and quoting

Earlier we saw that wildcard characters like `*` are substituted by the shell with portions of file names. We have just seen another form of subsitution: *`$var`* is replaced with the value in variable *`var`*. A third form of substitution is *command substitution*:

- `` `command` `` is replaced with whatever *command* sends to `stdout`

Note the quotes are back ticks (`` ` ``, the key usually to the left of `1`), not forward ticks (`'`, the key to the right of `;`).

Here is an example to demonstrate command substitution:

```
$ ls
file1 file2
$ echo $HOME
/home/cs114
$ cat file1
this is a line
$ cat file2
f* $HOME
$ echo `cat file1`
this is a line
$ echo `cat file2`
file1 file2 $HOME
```

As you can see, the result of command substitution is further substituted, but only wildcard expansion is performed. Variables are not substituted.

We also saw that wildcard characters can be passed to a program without substitution by escaping the character with a `\`, thus:

```
$ echo \*
*
$ echo *
file1 file2 file3
```

Escaping can also be done on other characters the shell treats specially. For example, the shell normally interprets whitespace on the command line as a separator between arguments to pass to a program.

```
$ echo a  b  c
a b c
$ echo a\ \ b\ \ c
a  b  c
```

Escaping with `\` can quickly get unwieldy. Instead, characters can be protected from substitution by *quoting* them.

- `"..."` – within double quotes, only variable and command substitutions occurs, not wildcard expansion.
- `'...'` – within single quotes, no substitutions occur whatsoever: no wildcard substitution, no variable subsitution, and no command substitution.

Thus:

```
$ x=d
$ echo a  b  c  $x
a b c d
$ echo "a  b  c  $x"
a  b  c  d
$ echo 'a  b  c  $x'
a  b  c  $x
$ echo "'a'"
'a'
$ echo '\'a\''
```

```
'a'
$ echo '"a"'
"a"
$ echo "\"a\""
"a"
$ echo "\'a\'"
\'a\'
```

Observe that within single quotes, single quotes need to be escaped. Similarly, within double quotes, double quotes need to be escaped.

Making thumbnails

To demonstrate quoting and substitution, below is a shell script that creates a thumbnail image for a JPEG image.

```
 1 #!/bin/sh
 2 # Create a thumbnail image from the file given on the command line.
 3 # For file foo.jpg, the script creates foo.thumb.jpg, which is
 4 # at most 200x200.
 5
 6 # Make sure the programs we need are in the path.
 7 PATH=/usr/bin:/bin:/usr/local/graphics/jpeg-6b/bin:/usr/local/pbm
 8
 9 jpg="$1"
10 thumb=`echo "$jpg" | sed 's/\.jpg$/.thumb.jpg/'`
11
12 echo "producing thumbnail $thumb from $jpg"
13
14 w=`rdjpgcom -verbose "$jpg" |
15     nawk '/^JPEG image is/ { sub("w$", "", $4); print $4 }'`
16 h=`rdjpgcom -verbose "$jpg" |
17     nawk '/^JPEG image is/ { sub("h,$", "", $6); print $6 }'`
18
19 echo "$jpg is $w by $h"
20
21 # Want a thumbnail that's at most 200*200, but preserves
22 # the $w*$h aspect ratio
23
24 # Find the maximum of $w and $h
25 max=$h
26 if [ $w -gt $h ]; then max=$w; fi
27
28 factor=`echo "scale=4; 200 / $max" | bc`
29
30 echo "scaling $jpg by $factor"
31 djpeg -pnm "$jpg" | pnmscale $factor | cjpeg > "$thumb"
```

Let's go through the script line-by-line. Since the script uses some image processing programs, line 7, sets the `PATH` so that these programs can be used without naming them with their full absolute path.

On line 9, the variable `jpg` is set to `$1`. The special `$1` variable contains the first command-line argument passed to the program. In this case, it is expected to be the name of a JPEG file. `$2` through `$9` contain the second through ninth command-line arguments. We'll discuss argument processing in more depth in a later lecture.

On line 10, `sed` is used to set the variable `thumb` to the name of the thumbnail file to create. If `$jpg` is `foo.jpg`, `$thumb` is `foo.thumb.jpg`.

Line 12 just echoes an informational message.

Lines 14 and 15 get the width and height of the JPEG respectively. The program `rdjpgcom` reads the comment field of the JPEG and with the `-verbose` option also prints out the size of the image. For example:

```
$ rdjpgcom -verbose ~/Pictures/desert.jpg
JPEG image is 967w * 750h, 3 color components, 8 bits per sample
JPEG process: Baseline
```

The output of `rdjpgcom` is passed to an `nawk` (`awk` is an enhanced version of `awk`) script, which extracts the width and height by grabbing the 4th and 6th fields, respectively, and trimming off the `w` and `h,`.

Lines 25 and 26 determine the maximum of `$w` and `$h`. Line 26 tests if `$w` is greater than (`-gt`) `$h` and, if so, sets `max` to `$w`.

Line 28 computes the scale factor, which is 200 over the maximum dimension. For example, if the image is 1200x1600, the scale factor is one-eighth: 0.125. The factor is computed using the calculator program `bc`. `bc` reads a program from `stdin`. In this case the program sets the `scale` to 4, which says that results should be printed to 4 decimal places, and then computes the scale factor `200 / $max`. Note that the program is double quoted so that `$max` is expanded, but the character "`;`" is not handled specially by the shell.

Finally, line 31, uses `djpeg` to convert the image to a PNM file, then uses `pnmscale` to rescale the image using the scaling factor we computed above, and finally, uses `cjpeg` to convert the PNM output by `pnmscale` back into a JPEG. `cjpeg` writes to `stdout`, which is redirected to the thumbnail file `$thumb`.

Counting lines of C code

The following shell script shows some of the limitations of `sed` and how they can be worked around.

This script counts lines of C (or C++, Java, C#) code, excluding comments and blank lines. The script processes files given on the command line or sent to the script from `stdin`.

```
$ cat loc
#!/bin/sh
# Count lines of C, C++, Java, or C# code, excluding comments and
blank lines.

sed -e 's!//.*$!!' $*     |    # Strip C++ // comments
tr '@%\012' '++@'         |    # Replace @ and % with + and newline
with @
sed -e 's!\*/!%!g'        |    # Replace */ with %
sed -e 's!/\*[^%]*%!!g'   |    # Strip /* ... %
tr @ '\012'               |    # Restore newlines
grep -v '^[      ]*$'     |    # Strip blank lines: [] contains a tab
and space
wc -l                          # Count the remaining lines
```

The script works as a pipeline of several commands. It first uses `sed` to remove C++ comments of the form:

```
// comment to end of line
```

This invocation of `sed` is passed the argument `$*`. `$*` is a special variable that expands to a list of all command-line arguments passed to the script.

Next, we have to remove multiline `/* ... */` comments. Since `sed` works line-by-line, the script uses `tr` to replace newline characters, written `\012` (012 is the octal representation of an ASCII newline

character), with a `@` character. We also replace `@` with `+` so that all `@`s can be later restored to newline. `%` characters are also replaced, since they will be treated specially in the next step.

Another limitation of `sed` is that it only supports basic regular expressions. In particular, `sed` does not support the `|` operator, so we can't write the (complicated) regex `/\*([^\*]|\*[^/])*\*/` to match a `/* ... */` comment. To work around this, we first replace all occurrences of `*/` with a `%`. Making the comment terminator a single character permits the `/* ... */` comments—now `/* ... %` comments—to be matched and removed with the regular expression `/\*[^%]*%`.

Next, we use `tr` again to restore newlines, then use `grep` to eliminate lines containing only whitespace, and finally use `wc` to count the number of lines remaining.