CA170: Week 5

Intro to Shell Programming

Intro

- Unix/Linux command-line - One-liner "programs" at the command prompt (e.g. prog piped to prog).
- Shell programming - Text file of multiple Unix/Linux commands (e.g. "if" condition then execute command, else other command).
- The combination of one-liner command-line plus multi-line Shell programs gives you a programmable User Interface, where you can quickly write short programs to automate repetitive tasks.
- Shell program = an interpreted program (i.e. not compiled). Also known as a "Shell script" or "batch file" of UNIX commands.
- Like .BAT files on Windows command line.

How to make a Shell Program

1. Put it in a directory that is in the PATH. In this course we will put it in $HOME/bin
2. It can have any extension (typically no extension).
3. Put valid UNIX commands in it.
4. Make it executable:

```
$ chmod +x file
```

5. Run it by typing its name:

```
$ file
$ file &
```

Alternative ways of working

1. Pass program as arg to shell:

```
$ sh prog
```

- Advantage: Does not have to be executable. Does not have to be in PATH.
- Disadvantage: Have to type "sh" all the time. Have to be in same directory (or else type more complicated path to program).

2. Use file extension:

```
$ prog.sh
```

- Advantage: Can easily see what the file is from a directory listing.
- Disadvantage: Have to type the .sh
- Could be used when no one ever types the name. (e.g. The script is only ever called by a program.)

Arguments and returns

- A Shell program is different to something typed on the command-line.
- It can have arguments, and can exit at any point.

```
$1      1st command-line argument
$2      2nd command-line argument
...

$*      all arguments

#       comment


exit        exit the Shell script
exit 0            exit with a return code that other progs can query

$?          return code of last prog executed
```

if, test and flow of control

- Conditional statements - link in notes
- Flow of control statements- link in notes
- Test - link in notes

```
# test if 1st argument = "0"

if test "$1" = "0"
then
 echo "yes"
else
 echo "no - first argument is $1"
fi
```

Sample Shell Program

norm - set permissions as open as possible

```
chmod  u+rwx,go+rx-w  $*
```

hide - as hidden as possible

```
chmod  u+rwx,go-rwx  $*
```

semihide - just open enough as needed for Web

- Directories

```
chmod  u+rwx,g-rwx,o+x-rw  $*
```

- Files (webpages , images, etc)

```
chmod  u+rwx,g-rwx,o+r-xw  $*
```

- "norm" could of course replace all 3 if you don't mind granting more access than strictly necessary.

rmifexists - silent repeated delete

- e.g. Want to be able to repeatedly run:

```
for i in $*
do
        if test -f $i
        then
         rm $i
        fi
done
```

- and not get error message if no *.bak files found.
- in fact, `rm -f` will do the job with no warnings
- Rm - link in notes

Recursive rm

- `rm -r` (recursive rm)
- Don't type:
- `rm -rf /`
- "Trying out some Deadly Linux Commands".
- Includes typing `rm -rf /` and other scary commands, including the hilarious: `mv / /dev/null`
- Link in notes + link to video

wipe - clean up editor backup files

```
rmifexists *%
rmifexists .*%

rmifexists  *~
rmifexists .*~

rmifexists  *.bak
rmifexists .*.bak

rmifexists  *.BAK
rmifexists .*.BAK
```

1. Easier than having to point-and-click each one. Especially if do this for multiple directories.
2. Safer than typing "rm *bak" every day. One day you will type "rm * bak"
3. In general, if you regularly type some command that would be dangerous if you make a typo, it would be better to debug it once and put it in a script and never type it directly again.

Command-line image processing

- We need libjpeg utilities.
- At DCU this is:
    - Installed on PCs in labs.
    - Not installed on student.computing.dcu.ie (ssh).
- To make 1/4 size versions of 10,000 JPEGs without ever opening an image editor (or doing any work):

```
for i in *jpg
do

 djpeg  -scale 1/4 -bmp $i  > temp.bmp

 cjpeg  temp.bmp            > small.$i

done
```

- JPEG needs to be decoded to a BMP (bitmap), then be re-sized, then re-coded back to JPEG.
- Can do this in one line, leaving out the temporary file: Pipe result of djpeg into input of cjpeg.
- Link to BMP in notes

Extract images from PDFs

- I was once given an archive of thousands of scanned historical images. They were all inside PDFs.
- I automatically extracted all the JPEGs from the PDFs as follows:

```
for i in *pdf                           # i = x.pdf
do

  x=`basename "$i" ".pdf"`         # get root filename x (without .pdf
bit)
```

```
        pdfimages  -j  $i  $x          # extracts images to x-nnn.jpg

    done
```

Command-line movie processing

- You can do command-line movie processing with ffmpeg.
- Your Shell script can bulk convert, split, join, rotate and resize thousands of videos. Without opening any window.
- ffmpeg command-line examples - link in notes

Sample Shell Scripts

Sample Shell script - filterbaks

- The following is to illustrate that .* files are only "hidden" by convention.
- We can pipe all our commands through a "filterbaks" program to make lots of other files also hidden by convention:
- Might hide text editor backup files (things like file%, file~, file.bak):
-

Filterbaks

- Might make it a separate file, called say "filterbaks":

```
grep -v "%$" | grep -v "~$" | grep -iv "\.bak$"

so we can reuse it in directory listing:

     ls -al $* | filterbaks

and in other progs:

      if test `echo $i | filterbaks`
      then
       ...
```

- Makes backup files invisible everywhere (until needed).
- Finally, it would be more efficient to replace 3 programs piped together with 1 program (with more complex arguments). (Why would this be more efficient?)
- We can do this using the "egrep" program, which can grep for boolean expressions. filterbaks can be rewritten as simply:

```
egrep -iv  "%$|~$|\.bak$"
```

- where, inside the string here, | means "OR".

d - my own ls script

```
ls -l $* | filterbaks
```

How to debug a program

- With the vast amount of sample code on the Internet, I often see students getting big chunks of sample code and trying vainly to get it to work. I also provide sample code which I ask students to modify. I have noticed students often do not have the right mindset when doing this.
- Here are a few simple tricks for how to debug a program:

1. Strip it down. Remove parts of it.
2. Get smaller parts working first.
3. Don't try to do it all at once.
4. You don't have to delete code. Just comment it out. Then slowly comment it back in.
5. Comment out lines of code:

```
# code
// code
```

6. A trick is to use tabs to make it easy to comment code out and in:

```
#     code
#     code
      code
#     code
```

7. Comment out blocks of code:

```
/*
code
code
*/
```

8. Insert an exit after reaching a certain stage (comments out everything below it):

```
code
code

      exit

code
Code
```

9. Look at variables half-way through:

```
echo $var

System.out.print(var);

console.log(var);
```

10. Build all programs in stages, testing each stage.
11. Slowly comment code back in.
12. Slowly move exit further down or remove it.
13. Slowly remove debug info.

- Debugging is not a mystery. Yes, there are fancy debugging and tracing tools.
- But half the time, a few well-chosen prints, exits and comment-outs are all you need to find the problem.
- Link in notes

Evolution of a Programmer

High School / Junior High

```
10 PRINT "HELLO WORLD"
20 END
```

First Year in College

```
program Hello(input, output)
 begin
    writeln('Hello World')
 end.
```

Senior Year in College

```
(defun hello
 (print
  (cons 'Hello (list 'World))))
```

New professional

```
#include
void main(void)
 {
  char *message[] = {"Hello ", "World"};
   int i;

   for(i = 0; i < 2; ++i)
     printf("%s", message[i]);
   printf("\n");
}
```

Seasoned professional

```
#include
#include
 class string
{
 private:
  int size;
   char *ptr;
 public:
   string() : size(0), ptr(new char('\0')) {}
   string(const string &s) : size(s.size)
  {
   ptr = new char[size + 1];
   strcpy(ptr, s.ptr);
   }
   ~string()
  {
   delete [] ptr;
   }
   friend ostream &operator <<(ostream &, const string &);
   string &operator=(const char *);
 };
ostream &operator<<(ostream &stream, const string &s)
{
   return(stream << s.ptr);
}
string &string::operator=(const char *chrs)
{
 if (this != &chrs)
  {
   delete [] ptr;
 size = strlen(chrs);
 ptr = new char[size + 1];
    strcpy(ptr, chrs);
 }
  return(*this);
}
 int main()
 {
   string str;
   str = "Hello World";
   cout << str << endl;
   return(0);
 }
```

Apprentice Hacker

```
#!/usr/local/bin/perl
$msg="Hello, world.\n";
if ($#ARGV >= 0) {
  while(defined($arg=shift(@ARGV))) {
    $outf = $arg;
    open(FILE, ">" . $outf) || die "Can't write $arg: $!\n";
    print (FILE $msg);
    close(FILE) || die "Can't close $arg: $!\n";
  }
}
else {
 print ($msg);
}
1;
```

Experienced Hacker

```
#include
#define S "Hello, World\n"
main(){exit(printf(S) == strlen(S) ? 0 : 1);}
```

Seasoned Hacker

```
% cc -o a.out ~/src/misc/hw/hw.c
% a.out
```

Guru Hacker

```
% cat
Hello, world.
^D
```

Junior Manager

```
10 PRINT "HELLO WORLD"
20 END
```

Middle Manager

```
mail -s "Hello, world." bob@b12
Bob, could you please write me a program that prints
"Hello, world."? I need it by tomorrow.
^D
```

Senior Manager

```
% zmail jim
I need a "Hello, world." program by this afternoon.
```

Chief Executive

```
% letter
letter: Command not found.
% mail
To: ^X ^F ^C
% help mail
help: Command not found.
% damn!
!: Event unrecognized
% logout
```