Contents of Lecture 3

- UNIX Shell programming
- UNIX commands

Why Shell Programming?

- A program written for a shell is called a shell script.
- Shell scripts are (almost always) interpreted (there is a company in the USA which sold shell-compilers but they now focus on selling C++ compilers instead).
- Shell programs have some advantages over C programs:
  - More convenient to write when dealing with files and text processing.
  - The building blocks of the shell are of course all the usual UNIX commands.
  - More portable.
- However, the shell is slower than compiled languages.

Different Shells

- There are a number of shells.
- Bourne shell is the original but lacked many features eg name completion.
- The csh and tcsh have different syntax but were more advanced.
- The Korn shell was written at Bell Labs as a superset of Bourne shell but with modern features.
- The GNU program Bourne Again Shell, or bash, is similar to Korn shell.
- We will focus on bash.

Bash as Login Shell

- Every user has a path to the login shell in the password file.
- When you login, and have bash as login shell, bash will process the following files:
  - `/etc/profile`
  - First found in `$HOME` of `.bash_profile`, `.bash_login`, `.profile`.
- When the login shell terminates, it will read the file `.bash_logout`.

Interactive Non-Login Shell

- An interactive shell is, of course, one which one types commands to.
- A non-interactive shell is one which is executing a shell script.
- An interactive shell which is not the login shell executes the file `.bashrc`.
- There is a file `/etc/bashrc` but it is not automatically read.
- To read it automatically, insert `source /etc/bashrc` in your `.bashrc`.

Non-Interactive Shell

- Non-interactive shells do not start with reading a specific file.
- If the environment variable `$BASH_ENV` (or `$ENV` if the bash was started as `/bin/sh`) contains a file name, then that file is read.
- The first argument to bash itself, contains the program name, so `echo $0` usually prints `bash`.

Source Builtin Command

- To ask the current shell to read some commands use the `source filename` command.
- You can use `.` instead of `source`.

Aliases and Noclobber

- UNIX commands perform their tasks without asking the user whether he/she really means what he/she just typed. This is very convenient (most of the time).
- For instance the `rm` command has an option `-i` to ask for confirmation before a file is removed.
- Sometimes people put the command `alias rm='rm -i'` in a bash start file.
- A similar feature is to use the command: `set -o noclobber` which avoids deleting an existing file with I/O redirection (eg `ls > x`).
- All such features should be avoided (in my opinion) since they just reduce productivity and make people think UNIX is a safe place.

I/O Redirection

Common directives include:

- `< file`: Use file as stdin.
- `> file`: Use file as stdout.
- `>> file`: Append output to file.
- `2> file`: Use file as stderr.
- `2>&1` : Close stderr and dup stdout to stderr.

Shell Script Basics

- The first line should contain the line `#!/bin/bash`
- To make the script executable, use `chmod a+x file`.
- A line comment is started with #.
- Commands are separated with newline or semicolon.
- Backslash continues a command on the next line.
- Parenthesis group commands and lets a new shell execute the group.

More about Parenthesis

- A subshell has its own shell variables such as current directory.
- The builtin `cd` does not read from stdin, so we can pipe as follows:
- We can now type

```
(cd ; ls) | (cd ~/Desktop; cat > ls-in-home)
```

Shell Variables

- Shell variables do not have to be declared — just assign to them:

```
$ a=unix
$ echo $a
$ b=wrong rm can have unexpected results such as disaster
$ c="wrong rm can have unexpected results such as disaster"
```

- The difference between the last two assignments is significant.
- A shell variable is by default local to the shell but can be exported to child processes using: `$ export a`.
- C/C++ programs get the value using `char* value = getenv("VAR");`.

Using Shell Variables

- Use a dollar sign before the name to get the value: `$HOME`.
- If you wish to concatenate a shell variable and a string, use

  `${VAR}suffix`

  without it you would get the wrong identifer

  `VARsuffix`

More about Using Shell Variables

- The value of `${var-thing}` is `$var` if `var` is defined, otherwise `thing` were `thing` is not expanded. Value of `var` is unchanged.
- The value of `${var=thing}` is `$var` if `var` is defined, otherwise `thing`; and `var` is set to `thing`.
- The value of `${var+thing}` is `thing` if `var` is defined, otherwise nothing.
- The value of `${var?message}` is `$var` if `var` is defined, otherwise a message is printed and the shell exits.

PS1 and PS2

- The prompts, `$` and `>` are called the primary and secondary prompts. These were the original values of these and they are stored in PS1 and PS2.
- For the `root` user, the prompt is #.
- It is possible to get a more informative prompt by using the escapes:

`\$`	# if root, otherwise dollar.
`\!`	Current history number (see below).
`\w`	Pathname of working directory.
`\W`	Basename of working directory.
`\h`	Hostname.
`\H`	Hostname including domain.
`\u`	User.
`\t`	24-hour time.
`\d`	Date.

Reexecuting Commands with a Builtin Editor

- To reexecute a command, use either the builtin editor (vi or emacs) as specified in your `.inputrc` file.
- `.inputrc` can contain eg `set editing-mode vi`.
- Using the editor is very convenient since you can change the command if it didn't work as expected. Simply hit `ESC` (for vi).
- This is a convenient way to experiment with new commands.

Reexecuting Commands with an Exclamation

- Commands available include:

`!!`	Reexecute most recent command.
`!`*n*	Reexecute command number *n*.
`!-`*n*	Reexecute the *n*th preceding command.
`!`*string*	Reexecute the most recent command starting with *string*.
`!?`*string*	Reexecute the most recent command containing with *string*.

- The last word on the previous command can be refered to as `!$`

```
$ ls -l f9.tex
$ vi !$
```

Quotation Marks

- There are three kinds of quotation marks:
- in a string enclosed by ": variables are expanded.
- in a string enclosed by ': variables are not expanded.
- the value of ‘`string`‘ is the stdout from executing `string` as a command and removing each trailing newline character:
  $ rm -rf ‘du -ks * | sort -n | awk ' { print $2 } '‘ # remove big file/directory
- You will find the last form useful during Lab 4.
- Note: the last form is equivalent to $(command).

Here Documents

- Sometimes it can be useful to provide the input to a script in the script file. The input is right "**here**".

```
$ cat phone
grep "$*" <<End
Office 046 222 9484
Mobile 0767 888 124
$X
End
```

- Above script contains both the command and the input.
- The variable `X` is expanded; suppress this behaviour by preceding `End` with a backslash on first line.

Functions

```
function fun()
{
        echo $1 # echo first argument
        echo $2 # echo second argument
}
```

- The keyword `function` is optional.
- A function must be declared before it can be used.
- A function can be used as if it was any other UNIX command, ie no parenthesis when the function is called (ie not even for passing arguments).

Simple Shell Syntax

- `a && b` executes `b` only if `a` succeeds (ie returns 0).
- `a || b` executes `b` only if `a` fails (ie returns nonzero).
- The following commands can cause harm if you run out of disk space during `tar`:

  ```
  $ tar cf dir.tar dir; rm -rf dir; bzip2 -9v dir.tar
  ```

- This is better:

  ```
  $ tar cf dir.tar dir && rm -rf dir && bzip2 -9v dir.tar
  ```

- Edit-compile-run without leaving the keyboard: `vi a.c && gcc a.c && a.out`

For Loops

- Iterate through certain files in your the current directory:

```
for x in *.c
do
        lpr $x
done
```

- or through all argumets passed to a script:

```
for x in $*
do
        lpr $x
done
```

More for Loops

- You can also iterate through a string:

```
a="x y z"
for s in $a
do
        echo $s
done
```

- Or simply a list:

```
for s in a b c
do
        echo $s
done
```

While and Until

```
while command
do
        body # do body while command returns true
done

until command
do
        body # do body while command returns false
done
```

If-Then-Else-Fi

```
if command
then
        then-commands
[else
        else-commands]
fi

if ! command
then
        then-commands
[else
        else-commands]
fi
```

Case

```
case word in
pattern1) commands;;
pattern2) commands;;
*)        commands;;
esac
```

- Nothing happens if no pattern matches: putting `*)` last makes a default.

cmp, diff, and ndiff

- `cmp` reports whether two files are equal.
- `diff` does the same but also shows how they differ.
- `ndiff` is a variant for which one can specify numerical differences which should be ignored.
- `ndiff` is not standard but easy to find.

cut

- `cut` cuts out characters from each line of stdin
- `ls -l | cut -c2-10` prints the rwx-flags of the files.
- The first character on a line is `c1`.
- Multiple ranges can be specified:
- `ls -l | cut -c2-10 -c51-55` also prints five characters from the file name.

find

- Example: `find . -name '*.c'`. The output will be a list of files (with full path) with suffix `c`.
- We can feed that list to `wc` using:
  ```
  wc `find . -name '*.java'`
  ```
- The default action is to print the file name.
- A number of criterions can be specified, including
  1. `-anewer filename` selects files newer than `filename`.
  2. `-type type` selects files of type `type` which is one of *b*,*c*,*d*,*f*, *l*, *p*, or *s* (with the same meaning as printed by `ls -l`: block special file (eg disk), character special file (eg usb port), directory, ordinary file, symbolic link, name pipe, or socket).

cleanfiles

```
find . -name *.tac.??? -exec rm '{}' \;
find . -name *.pr -exec rm '{}' \;
find . -name cmd.gdb -exec rm '{}' \;
find . -name *.ps -exec rm '{}' \;
find . -name *.dot -exec rm '{}' \;
find . -name *.aux -exec rm '{}' \;
find . -name *.o -exec rm '{}' \;
find . -name out -exec rm '{}' \;
find . -name x -exec rm '{}' \;
find . -name y -exec rm '{}' \;
find . -name a.out -exec rm '{}' \;
find . -name cachegrind.out.* -exec rm '{}' \;
```

awk

- Stands for Aho (from the Dragonbook), Weinberger (from `hashpjw` in the Dragonbook), and Kernighan (the K in K&R C).
- Each line of input is separated into fields and are denoted `$1`, `$2`, ...
- Assume a variable is called `X` and has value 2. Then `$X` refers to the second field.
- The entire line is `$0`, number of fields on a line is denoted `NF`, and line number is `NR`.
- Each line in an awk program has a pattern and an action.
- If a line in the input matches the pattern, the action is executed.

Example awk programs

```
$ awk '{ print $1, $5; }'          # print first and fifth item.
$ awk '$1 > 10 { print $1, $2; }' # print first two items if $1 is > 10.
$ awk 'NR == 10'                   # print tenth line.
$ awk 'NF > 4'                     # print each line with > 4 fields.
$ awk 'NF > 0 '                    # print each nonempty line.
$ awk '$NF > 4 '                   # print each line with last field > 4.
$ awk '/abc/ '                     # print each line containing abc.
$ awk '/abc/ { n = n + 1; }\
        END { print n;}'           # print number of lines containing abc.
$ awk 'length($0) > 80'            # print each line longer than 80 bytes.
```

- The END pattern matches at EOF. There is also a BEGIN pattern which is matched before the first character is read.

head and tail

- `head` prints the first 10 lines of a file (or stdin).
- `head -100` prints the first 100 lines of a file (or stdin).
- `tail` prints the last 10 lines of a file (or stdin).
- `tail -100` prints the last 100 lines of a file (or stdin).
- `tail -f file` like normal tail but at EOF waits for more data.

od

- Octal dump
- `od file` dumps the file contents on stdout in as octal numbers.
- `od -c file` prints file as characters.
- `od -x file` prints file as hex numbers.

sed

- sed stands for stream editor.
- It can be useful for eg changing prefixes in a Yacc generated parser:
- `sed 's/yydebug/pp_debug/g' y.tab.c > tmp;mv tmp y.tab.c`

grep

- Grep searches for a pattern in files.
- GNU grep has the useful `-r` option which traverses directories.
- In basic regular expressions ?, +, braces, parentheses and bar (ie |) have no special meaning. Backslash them to get that.
- In extended regular expressions, enabled with -E, above characters are special. More about that on next slide.

```
$ grep abc           # matches line with abc.
$ grep -e '[abc]'    # matches line with any of a, b, or c.
$ grep -e '[^abc]'   # matches line with none of a, b, or c.
$ grep -e '[^ab-d]' # matches line with none of a, b, c, or d.
$ grep ab*c          # matches line with ac, abc, abbbbbc.
```

grep -E

```
$ grep -E -e 'a|b'       # matches line with a or b.
$ grep -E -e 'a|bc'      # matches line with a or bc.
$ grep -E -e '(a|b)c'    # matches line with a or b, followed by c.
$ grep -E -e '(a|b)?c'   # ? denotes optional item.
$ grep -E -e '(a|b)+c'   # + denotes at least once.
$ grep -E -e '(a|b)*c'   # + denotes zero or more.
$ grep -E -e '(a|b){4}c' # {4} matches pattern four times.
```

- Without -E use backslash before above metacharacters.
- Without ' the shell will try to setup a pipe.

sort and uniq

- `sort file` sorts a file alphabetically.
- `sort -n file` sorts a file numerically.
- `uniq` removes duplicates line if found in sequence

svi

```
vi -c /$1 `egrep -e $1 *.[ch] */*.[ych] |
        awk -F: ' { print $1; } ' |
        uniq |
        sort`
```

- What does this script do?