COMP2303/COMP7306 Network and OS Principles

Week 4 - Tuesday

Version control - svn

School of Information Technology and Electrical Engineering
The University of Queensland

This Week

- Lectures
  - Version control
  - Unix shell
  - Shell scripts
- Pracs
  - Finish A1
  - Practice Debugging

Outline (svn)

- High-level
  - Concepts
  - Operations
- Subversion
  - Demo
- Subversion versus DIY
- Shells later

Version Control

- **Version control (source control) –** Tools to manage changes during a project's development.
  - Many systems. Eg cvs, subversion(svn), git, bazaar, source safe, mercurial, ...
  - Lots of arguments about the best tool or method.
- The main concepts transfer between tools.
- We are only focusing on centralized VC.
- See http://svnbook.red-bean.com.

Concepts

- Repository – stores the history of the project. You do not modify this directly.
- Working copy – a copy of the files in the project where normal programming activity happens. (Could be on a different computer to the repository).
- *State** – the contents of all the files in the project.

Operations

- Single user
  - checkout – *I'd like a working copy.*
  - commit – *remember this state.*
  - add/remove/rename
  - diff/status – *what have I changed?*
  - clean copy/revert – *put it back the way it was.*
  - tag – assign a label to a state.
    - Eg: ass1 complete, release_V1

Operations

- If multiple users are committing to the same repository, there may be commits which you don't know about.
- update – *Bring my working copy up to date with changes from the repository.*
  - *What if I've made changes as well?*
  - *Intelligent merging rather than blind copy.*

Operations

- blame/praise/annotate – *who changed that line last (and in what version)?*
- Branching – make a separate line of development within the repository. Commits to a branch do not affect other branches or the "trunk".
  - Useful for experiments or when making large changes without disturbing people until they are done.

Subversion (svn)

- svn is a replacement for CVS.
- svn is self documenting.
  svn help
  svn help *command*
- svn checkout *URL working-dir*
  - URL – where to find the repository.
    - https://example.com/svn/project/trunk
- A working copy has hidden audit info in .svn directories.

Svn demo

- svn status
- svn diff
- svn revert
- svn help
- svn commit
  - Editor for log messages (or –m)

Version numbers

- In svn, the repository as a whole has a version number. Each time a commit is made the version number goes up.
- cvs has a more complicated system.

Svn demo

- svn add *files*
- svn move *oldname newname*
- svn mkdir *dirname*
- svn rm
  - Note: These operations need to be committed.
- svn status

COMP2303 7306

Svn demo

- svn status –u
- Dealing with conflict.
  - svn resolve – "I have investigated and fixed the problem."
  - svn revert – "forget about my changes"

COMP2303 7306

Svn vs DIY

- How does svn compare with doing your own backups?
  - You can view the project in any previous committed state.
    - Backup systems might only be able to produce the latest state. Or, they thin out older backups.
  - Efficiency – (for text formats) svn stores differences between files rather than a whole new copy.

COMP2303 7306

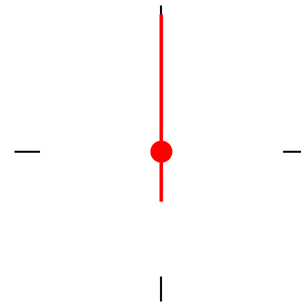
Svn vs DIY

  - Times when backups/snapshots are made may not coincide with states you wish to preserve.
  - How do you manage multiple developers?

COMP2303 7306

Short Break

- Stand up and stretch



COMP2303 7306

Outline (shells)

- UNIX Shell + Shell Scripts
  - Based on Glass & Ables – mostly chapter 4 but also 5 and 8
- Useful UNIX – shell aliases

COMP2303 7306

What is a Shell?

- Interface between the user and the operating system
- Provides
  - Input/output redirection, pipes
  - Wildcards
  - Job/process management (e.g. background jobs)
  - Command history
  - Command line editing
  - Some built in commands/functions
  - Scripting functionality
  - ... plus more

What Shells are There?

- Many
  - Thompson Shell (sh) – the original UNIX shell
  - Bourne Shell (sh) – from UNIX v7 (1977)
  - Bourne-Again shell (bash) – superset of sh
    - What you're probably using on agave
  - Korn Shell (ksh)
  - Z shell (zsh)
  - C shell (csh)
  - TENEX C shell (tcsh)
  - Scheme shell (scsh)
  - ...

What Does a Shell do?

- When invoked (e.g. by login process or manually)
  1. Reads startup file(s) and initialises
  2. Displays a prompt and waits for a user command
  3. If user indicates end-of-input (often Ctrl-D), shell terminates, otherwise executes user command and returns to step 2
- Shell scripts are similar, except commands come from a text file

Executable Programs vs Built-in Commands

- Commands can be either
  - Shell **built-in commands** (i.e. shell does something, no external process started), or
  - Separate executables (i.e. shell starts up an external process)
- All shells have this concept but the built-in commands available in each shell differ

Command Examples

- ls, sort, vim, pico, gcc, indent, which (external)
- cd, alias, type (builtin)
- echo, pwd, printf (either)
- Bash built-in command `type` will tell you which type of command
- Bash built-in command `help` will list the built-in commands (and can be used to get help on them)
- Built-ins executed in preference to external commands unless
  - disable built-in, or
  - give full path to external command

Variables

- Shell has two kinds of variables
  - Local (or shell) variables
    - Used only by the shell
  - Environment variables
    - Values passed to child processes
- Variables are strings
- Values accessed using `$varname` notation
  - e.g. `echo $HOME $USER $SHELL`

Predefined Environment Variables

Include:

- **HOME** – full pathname of home directory
- **PATH** – colon separated list of pathnames to search for commands
- **USER** – your username
- **SHELL** – full pathname of your login shell

COMP2303 7306

Defining a Variable

variableName=value

- No spaces around the equals sign
- Can change a shell variable to an environment variable using built-in command `export` , e.g.

```
courseCode=COMP2303
export courseCode
```

- Example to be given in class
- No need to declare variables before use
- Different shells use different syntaxes for defining variables (above is for Bourne, Bash)

COMP2303 7306

Break

COMP2303 7306

Useful UNIX

- Shell aliases
- To be demonstrated in class

COMP2303 7306

Metacharacters

- Some characters mean special things to the shell
- When you type a command, these **metacharacters** are processed before the command is executed
- If you don't want the special treatment, use backslash before the character or quote appropriately (discussed later)

COMP2303 7306

Metacharacters (cont.)

- Wildcards (for filename matching)
  - `*` = zero or more characters
  - `?` = any single character
- Comment
  - `#` = start of comment (goes till end of line)
- Running commands
  - `&` = run command in background
  - `;` = used to separate commands
  - `` `command` `` = substitute result of running command
- Variable substitution
  - `$varname` = substitute value of variable

COMP2303 7306

Metacharacters (cont.)

- Subshell
  - `( … commands …)` = execute commands in a sub-shell
- Conditional sequences
  - `||` = execute command if previous command failed
  - `&&` = execute command if previous one succeeded
- Command "succeeds" if returns zero exit status; "fails" if returns non-zero

Metacharacters (cont.)

- Redirection & Pipes
  - | = pipe, output of one program sent to the input of the next
  - > = send standard output to a file
  - < = read standard input from a file
  - >> = append standard output to a file
- There are other metacharacters also (and can vary by shell)

Examples

- To be presented in class
  - Wildcards (* ?)
  - Redirection (< > >>)
    - /dev/null
  - Pipes (|)
  - Command sequences (; || &&)
  - Subshell (())
  - Command substitution (`)
  - Background processing (&)

Shell Scripts

- Shell script = series of commands in a regular text file
- Can be made executable and executed like a regular command:

```
chmod +x script-file-name
./script-file-name
```
(if it's in current directory)

- How does the system know which shell to use? – depends on first line of the shell script
  - `#` - use the current shell
  - `#!pathName` – use the shell with the specified path
    - e.g. `#!/opt/local/bin/bash`
  - anything else, use the Bourne Shell (`/bin/sh`)

Built-in Shell Variables

All shells support:

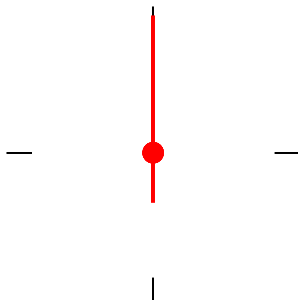
- **$$** - process ID of the shell
- **$0** – name of the shell script (if applicable)
- **$1** ... **$9** – command line arguments (if applicable)
- **$*** – all the command line arguments

Bourne shell/Bash support (amongst others):

- **$#** - number of command line arguments
  - excludes command name
- **$?** – exit status of last command
- **$!** – process ID of last background command

Short Break

- Stand up and stretch



Quoting

- Sometimes want to stop the shell replacing metacharacters
- Single quotes – inhibit wildcard replacement, variable substitution, command substitution
- Double quotes – inhibit wildcard replacement only
- Can also use backslashes
- Examples in class...

COMP2303 7306

Startup Files

- Files read at startup vary
  - by shell (different shells use different files)
  - by mode
    - login shell
    - interactive shell
    - non-interactive (shell script)
- Files contain commands which are executed (or **sourced**)
  - Not a separate process – commands are executed within the current shell – just like you'd typed them in
  - To source a file within Bash, use either:
    - . filename
    - source file

COMP2303 7306

Bash Startup

- Login Shell
  - `/etc/profile` (system wide settings)
  - `~/.bash_profile` OR `~/.bash_login` OR `~/.profile`
  - (`~/.bash_logout` executed on exit)
- Interactive Shell
  - `~/.bashrc`
- Shell Script
  - Looks for file named in `BASH_ENV` environment variable

- On agave
  - `/etc/profile` executes
    - `/etc/profile.agave`
    - `/etc/bash.env` which executes `~/.bashrc` if it exists

COMP2303 7306

Other Shell Features

- To be discussed/demonstrated in class

COMP2303 7306

Shell Arithmetic

- expr command
  - Usage: **`expr epxression`**
- Integer expressions using operators
  `*  /  %  +  -  =  >  >=  <  <=  !=  &  |`
- Need spaces between all arguments
- Parentheses allowed also
- Need to escape (e.g. with backslash) characters which mean something to the shell
- Examples:
  - **`` x=`expr \( 2 + 3 \) * 5` ``**
  - **`` i=`expr $i + 1` ``**
- String expressions possible also. See **`info expr`** for details

COMP2303 7306

Conditional Expressions

- test command
  - Usage: **`test expression`**
- Exits with status 0 (success) if expression is true, non-zero otherwise
- Can be used to compare integers or strings and also test files
- Examples:
  - **`test -f filename`** True if filename exists as a file
  - **`test -d filename`** True if filename is a directory
  - **`test string`** String is not null
- More examples to be given in class

COMP2303 7306

Bash Variations on `expr` and `test`

- Bash supports **`$((…))`** notation for arithmetic expressions
  - evaluated by shell, not by separate executable (**`expr`**)
- Bash supports **`[ … ]`** notation for conditional expressions
  - **`[`** is the same as **`test`** (shell builtin)\
- Examples

Control Structures

- **for name** [**in** *word…*]
  do
  commands…
  done

Means this part is optional

- variable name is assigned each of the the words in turn
- If words omitted, uses script arguments $1 $2 ...
- Examples...

Control Structures (cont)

- **if** *commands1…*
  then
  commands2…
  elif *commands3…*
  then
  commands4…
  else
  commands5…
  fi
- **elif** and **else** branches are optional
  - Can have multiple **elif** branches
- If last command in *commands1…* succeeds (exit status 0) then *commands2…* executed, etc
- Example (commands are often test expressions)

Control Structures (cont)

- **while** *commands1…*
  do
  commands2…
  done
- *commands1…* commands executed and if last command has exit status 0, then *commands2…* executed – repeat until *commands1…* return non-zero exit status
- Example ...

Other Control Structures

- case...
- until ... do... done
- trap