CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Spring 2008

Lecture 2— Processes, Users, Shell Special Characters, Emacs

Where are we

It's like we started over using the computer from scratch.

And all we can do is run dinky programs at the command-line.

But we are learning a *model* (the system is files, processes, and users) and a powerful way to *control* it (the shell).

If we get the model right, hopefully we can learn lots of details quickly.

Today:

- The rest of the model briefly: Processes and Users
- More programs (`ps`, `chmod`, `kill`, ...)
- Special shell characters (`*`, `~`, ...)
- Text editing (particularly emacs)

Announcements &c.

- For file and directory operations (rm, cp, mv, ...), use man or Pocket Guide pages 37–55 (or maybe 37–70)
- Homework 1A is posted, due **Monday** before class. You can do it after today.
- There will be a Homework 1B (basic shell scripting) handed out by Friday.
- Bash reference manual in html linked from course webpage, or use `info bash` for the `info` version.
- Logistics 1: assignment deadlines
  - What time? (2 pm before class, 10 pm, other?)
  - What day(s)?
- Logistics 2: Office hours blacklist

Users

- There is one file-system, one operating system, (often) one CPU, and multiple users.
- `whoami`
- `ls -l` and `chmod` (permissions), `quota` (limits)
  - Make your homework unreadable by others!
- `/etc/passwd` (or equivalent) guides the *login* program:
  - Correct username and password
  - Home directory
  - Which shell to open (pass it the home directory)
  - The shell then takes over, with *startup scripts* (e.g., `.bash_login`). (`ls -a`)
- one "superuser" a.k.a. *root*. (Change passwords, halt machine, ...)

Processes

- A running program is called a *process*. An *application* (e.g., emacs), may be running as 0, 1, or 57 processes at any time.
- The shell runs a program by "launching a process" waiting for it to finish, and giving you your prompt back.
  - What you want for `ls`, but not for `emacs`.
  - `&`, `jobs`, `fg`, `bg`, `kill`
  - `ps`, `top`
- A running shell is just a process that kills itself when interpreting the `exit` command.
- (Apologies for aggressive vocabulary, but we're stuck with it for now.)

That's most of a running system

- File-system, users, processes
- The operating system manages these
- Processes can do I/O, change files, launch other processes.
- Other things: Input/Output devices (monitor, keyboard, network)
- GUIs don't change any of this, but they do hide it a bit.

Now: Back to the shell...

Complicating the shell

So far, our view of the shell is the barest minimum:

- *builtins* affect subsequent interpretations. New: `source`
- Otherwise, the first "word" is a program run with the other "words" passed as arguments.
  - Programs interpret arguments arbitrarily, but conventions exist.

But you want (and `bash` has) so much more:

- Filename metacharacters
- Pipes and Redirections (redirecting I/O from and to files)
- Command-line editing and history access
- Shell and environment variables
- Programming Constructs (ifs, loops, arrays, expressions, ...)

All together, a very powerful feature set, but awfully unelegant.

Filename metacharacters

Much happens to a command-line to turn it into a "call program with arguments" (or "invoke builtin").

Certain characters can *expand* into (potentially) multiple filenames:

- `~foo` – home directory of user `foo`
- `~` – current user's home directory (same as `~$user` or `` `whoami` ``).
- `*` (by itself) – all files in current directory
- `*` – *match* 0 or more filename characters
- `?` – *match* 1 filename character
- `[abc]`, `[a-E]`, `[^a]`, ... more matching

Remember, this happens *before* deciding what to pass to a program.

Filename metacharacters: why

- Manually, you use them all the time to *save typing*.
- In scripts, you use them for *flexibility*. Example: You do not know what files will be in a directory, but you can still do: `cat *` (though a better script would skip directories).

But what if it's not what you want? Use quoting (`"*"`) or escaping (`\*`).

The rules on what needs escaping where are *very* arcane.

A way to experiment: `echo`

- `echo` *args...* copies its arguments to standard output *after* expanding metacharacters.

History

- The `history` builtin
- The `!` special character
  - `!!`, `!n`, `!abc`, ...
  - Can add, substitute, etc.

This is really for fast manual use; not so useful in scripts.

Where are we

Features of the `bash` "language":

1. builtins
2. program execution
3. filename expansion (Pocket Guide 22–23)
4. history

---

5. command-line editing
6. shell and environment variables
7. programming constructs

But file editing is too useful to put off... so a detour to `emacs` (which shares some editing commands with `bash`)

What is emacs?

A programmable, extensible text editor, with lots of goodies for programmers.

Not a full-blown IDE. Much "heavier weight" than `vi`.

Top-6 commands:

- `C-g`
- `C-x C-f`
- `C-x C-s`, `C-x C-w`
- `C-x C-c`
- `C-x b`
- `C-k`, `C-w`, `C-y`, ...

Take the emacs tutorial to get the hang of the basics.

Customizable with `elisp` (starting with your `.emacs`).

Command-line editing

Lots of control-characters for moving around and editing the command-line. (Pocket Guide page 28, emacs-help, and Bash reference manual Section 8.4.)

They make no sense in scripts.

Gotcha: `C-s` is a strange one (stops displaying output until `C-q`, but input does get executed).

Good news: many of the control characters have the same meaning in `emacs` (and `bash` has a `vi` "mode" too).

Bash startup files

Bash reads (sources) specific files when it starts up. Put commands here that you want to execute every time you run bash.

Which file gets read depends on whether bash is starting as a "login shell" or not

- Login shell: `~/.bash_profile` (or others if this is not found)
- Non-login shell: `~/.bashrc` (or others if not found)

Suggestion: Include the following in your `.bash_profile` file so the commands in `.bashrc` will execute regardless of how the shell starts up

```
if [ -f ~/.bashrc ]; then source ~/.bashrc; fi
```

(This *will* make sense shortly — promise!)

Putting it all together: Java

Java is a programming language; you can write and run programs in various environments.

The `javac` and `java` programs "compile" and "run" Java programs and `emacs` has a decent `Java` mode.

So we can write Java files in emacs, and use the shell to run the program and pass arguments.

(The Java program takes the class whose main should be run as its first argument and gives it the remaining arguments.)

Summary

As promised, we are flying through this stuff!

- Your computing environment has files, processes, users, a shell, and programs (including `emacs`).
- Lots of small programs for files, permissions, manuals, etc.
- The shell has strange rules for interpreting command-lines. So far:
  - Filename expansion
  - History expansion
- The shell has lots of ways to customize/automate. So far:
  - `alias` and `source`
  - run `.bash_login` or `.bashrc` when shell starts.
    - (or `.bash_profile` – look up the differences)

Next: I/O Redirection, Shell Programming