Bash Basics 1: Intro & Commands

These slides introduce the basics of working with **Bash**:

1. **Intro & Commands**
   - **shells and CLI**
   - **GNU project**
   - **command interpreter**
   - **types of commands in Bash**
   - **commands syntax**
2. Expansions & Metachars
3. Variables & Quoting
4. Interactive Use
5. Redirections & Locale

Bash

A **shell** is a piece of software that provides a **command line interface** (**CLI**).

A CLI allows users to type commands to be run by the OS.

The default Linux **shell** is **Bash** (**Bourne Again SHell**).

It is a GNU extension of the **Bourne shell** (`sh`), one of the first UNIX shells.

Bash is an excellent shell both for **interactive** use and for **shell scripting** (programming).

The Bash executable is typically: `/bin/bash`

Linux systems do not have a true Bourne shell, but rather use Bash in "legacy mode" (when invoked as `sh`).

Source Code and Further Information

Bash is open source software, provided by the GNU project:
`http://www.gnu.org/software/bash`

The *Bash Reference Manual* can be found at:
`http://www.gnu.org/software/bash/manual`

The *Bash Prompt HOWTO* can be found at:
`http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO`

Command Interpreter

A shell does not simply take the line you type and run it.

Bash first "interprets" the line as follows:

1. the input line is broken into **tokens** ("**words**" + "**operators**")
2. **aliases** are applied (i.e., **macros**, possibly rewriting commands)
3. the tokens are *parsed* to identify individual **commands**
4. a sequence of **shell expansions** are performed
   (possibly rewriting commands)
5. I/O **redirections** are performed

After this processing, the command(s) that result are **executed** (each simple command is run in a separate **subprocess**).

Bash *waits* for all command(s) to terminate before prompting for the next input line.

Commands

Bash supports several types of **commands**:

- **simple commands** – a single command/executable
- **pipelines** – sequence of simple commands run *in parallel*, the *output* of each command becomes the *input* of the next
- **lists** – multiple simple commands or pipelines, run *sequentially*, and possibly *conditionally* (based on success)
- **compound commands** – conditional, looping, and grouping commands

Commands (contd.)

A command line is typically terminated by typing *enter/return*, which indicates **newline/end-of-line**.

Unlike *C-family languages*, in Bash the *newline is meaningful*, so you don't need to put semicolons at the end of every Bash line.

A command line can also be terminated with a "**&**" (*ampersand*) before typing enter/return.

This causes the command be run "in the **background**," which means it is run **asynchronously** with the shell, in a subprocess.

"**Backgrounding a command**" is often done when the command starts a GUI program, since this will allow the shell to continue to be used (the shell will immediately prompt for further input).

Simple Command Syntax

The standard syntax for a **simple command** is:

```
command [option...] [arg...]
```

`command` is either:

- the *name* of the command; or
- a **pathname** (**path**) of an executable.

If only a command *name* is given, the shell locates the executable by looking through the directories in the **shell search path** for a file with the specified name, which is executable by the user.

The shell search path is maintained in the **PATH environment variable**, as a *colon*-separated list of directory pathnames:

`printenv PATH` ⇒ `/usr/bin:/bin:/usr/local/bin:...`

Simple Command Syntax (contd.)

`[option...]` indicates that one or more **options** may be specified after the command, but that options are *optional*.

`[arg...]` indicates that one or more **command arguments** (also called **operands**) can be supplied after any options, but that arguments may be optional.

If supplied, command **options** are typically in one of *two forms*:

- *old/short style* – *single letters*, prefixed with a *single dash* (`-`)
- *new/long style* – words, prefixed with a *double dash* (`--`)

Examples:

```
ls -a -d *
ls --all --directory *
```

Simple Command Syntax (contd.)

With old style options:

- multiple options may be listed together:
  ```
  ls -ad *
  ```
- if an option requires an **argument**, the argument may be put right after the option letter or separated by whitespace:
  ```
  ls -adw80 *
  ```

With new style options:

- if an option requires an argument, it is done using "=":
  ```
  ls --all --directory --width=80 *
  ```

Old and new options may be mixed:

```
ls -ad --width=80 *
```

Dash or Hyphen or Minus Sign?

What should the ASCII character 0x2D (which has a standard printed representation of "-") be called?

The term *"dash"* was used above, but many people call it *hyphen* and some *minus sign*.

SUS does refer to it as *hyphen*, but mainly as "the `'-'` character."

In English, there is a difference between a *dash* and a *hyphen*, both in purpose and in typical printed representation.

However, when using 0x2D to introduce options, the character is functioning neither as an English dash nor hyphen (and most certainly not as an arithmetic operator).

We tend to prefer the single syllable "dash," but these are not English dashes any more than those are "periods" in "`www.cs.siu.edu`".

Pipelines

One of the most powerful aspects of Linux/UNIX shells, is the ability to construct unique, complex commands by connecting together *sequences of simple commands* into a **pipeline**.

This is accomplished by using **pipes**, a Linux/UNIX mechanism for **interprocess communication**.

The shell pipeline symbol is: | (the *vertical bar*).

The basic format of a shell pipeline command is:

```
simple_command1 | simple_command2
```

What this does is to setup a pipe that passes the *output* from `simple_command1` to become the *input* to `simple_command2`.

Pipelines (contd.)

More precisely, for `simple_command1 | simple_command2` :

- the shell creates a **pipe** object
- one **subprocess** is created to run `simple_command1`
- in that subprocess, **standard output** is **redirected** to the **write end** of the pipe
- a second **subprocess** is created to run `simple_command2`
- in that subprocess, **standard input** is **redirected** to the **read end** of the pipe
- the two commands are run **concurrently** in the two subprocesses, while the shell waits

Pipelines (contd.)

For a pipe to work, the individual commands must *read from* **standard input** and *write to* **standard output**.

Commands of this type are called **filters**.

Many standard filter commands are available for use:
`cut`, `grep`, `head`, `nl`, `sort`, `split`, `tail`, `tr`, etc.

Multiple pipelines may be used to string together more than two filter commands.

In this way, complex and unique functionality can easily be constructed by users from multiple relatively simple programs.

Pipelines (contd.)

Pipeline examples:

- `grep printf lab2.c | wc -l` (count `printf`'s in `lab2.c`)
- `ls -l *.c | grep Jun | wc -l` (count C files modified in June)

Several commands can function as “normal commands” or as filters, depending on how they are called:

- `grep [OPTIONS] PATTERN [FILE...]`
- reads from `FILE` argument(s) if any are given, else reads from **standard input**

Lists

A **list** command consists of multiple simple commands or pipelines separated by one of:

- `;` (semicolon) – commands are run *sequentially*
- `&&` (double ampersand) – each command is run only if the previous *“succeeded”*
- `||` (double vertical bar) – each command is run only if the previous *“failed”*

Success/failure refers to the **exit status** of a command.

“`&&`” is effectively *“short-circuit and”* while “`||`” is effectively *“short-circuit or”*.

Compound Commands

Compound commands is the general name for Bash' **grouping commands** and **control constructs**.

Grouping commands simply allow a **list** to be treated as a *unit* (for redirection):

- (*list*) – run *list* in subshell
- { *list*; } – run *list* in current shell context

Control Constructs

Shells typically provide a variety of **control constructs**, though syntax varies among shells.

Bash **conditional** constructs:
if-then, if-then-else, if-then-elseif, case (switch)

Bash **looping** constructs:
while, until, for, map (through a list)

Control constructs are mainly used in **shell scripts** (shell language programs).

However they are often useful in interactive commands:

```
  for f in *.txt; do mv "$f" "${f%txt}text"; done
```

(change the extensions of all `.txt` files in the CWD to `.text`, something that cannot be done with a single `mv` command)

Shell Scripts

A **shell script** is a program written in a shell language and placed into an executable file.

Scripts are often used to automate tasks that involve a sequence of commands (so can be run by cron).

E.g., convert mp3's to wav's in some directory:

```
#!/bin/bash

cd "$1"  #cd to directory given as first command-line argument

for f in *.{mp3,MP3}; do
  lame --decode "$f" "${f%.*}.wav"
done

exit 0
```