CSE 303
Concepts and Tools for Software Development

Magdalena Balazinska
Winter 2010
Lecture 3 – I/O Redirection, Shell Scripts

Where We Are

Last two lectures

- A simple view of the system: files, users, processes, shell

- Lots of small useful programs; more to come

Today

- Introduction to emacs

- Input/Output redirection

- Combining commands

- Shell scripts

Introduction to emacs

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

- Not a full-blown IDE

- Much "heavier weight" than vi

Basic Emacs Commands

`C-x C-f`: open file

`C-x 5 f`: open file in new window

`C-x C-s`: save

`C-x C-w`: save as

`C-x C-c`: exit

`C-x b`: switch to another buffer

`C-g`: cancel partially typed command

Additional Useful Commands

- `C-k`: cut line

- `C-y`: paste line

- `M-/`: auto-complete (`M` means `ESC` key)

- `C-x 2`: split frame in two (`C-x 0`)

- Fancier copy-paste exists

- Many fancy commands: auto-indent, comment-region or uncomment-region

- Color customization: "Customizing Faces"

Command Line Editing

- Can use a lot of same commands as emacs

- More info in the Linux Pocket Guide (p28)

- Note: you will not be evaluated on command line editing. It's just for you.

Program Inputs and Outputs

- What we already know...

- Program takes array of strings as argument

    - Some of these arguments can be options

- Program returns an integer

    - Convention: 0 for success, non-zero for failure

    - Previous command's exist status is in `$?`

Program Inputs and Outputs

- The shell also sets up 3 "streams" of data for the program

- stdin is an input stream with file descriptor 0
  - Standard input, default keyboard

- stdout is an output stream with file descriptor 1
  - Standard output, default shell window

- stderr is an output stream with file descriptor 2
  - Standard error, default shell window
  - Normally used for error messages

Input/Output Redirection

- Using special characters we can tell the shell to use files instead of the keyboard and screen (online Bash manual section 3.6)

- Redirect input: `cmd < file`

- Redirect output, overwrite file: `cmd > file`

- Redirect output, append file: `cmd >> file`

- Redirect error output: `cmd 2> file`

- Redirect both stdout, stderr: `cmd &> file`

I/O Redirection Examples

Sample commands (output not shown)

```
man ls > manual-page.txt

man idonotexit > manual-page.txt

man idonotexit 2> manual-page.txt

man ls > manual-page.txt 2>&1

man idonotexist > manual-page.txt 2>&1

man ls &> manual-page.txt

man ls >> manual-page.txt

history > my-history
```

Pipes

```
cmd1 | cmd2
```

- Change the stdout of cmd1 and the stdin of cmd2 to be the same new stream

- Very powerful idea

  - Can combine many small programs into more complex programs!

  - `grep --help | less`

  - `history | grep man`

Combining Commands

- `cmd1; cmd2` **(sequence)**

- `cmd1 || cmd2` **(or)**

  - Using the integer return value ("exist status")

  - Execution of commands stops after first success

- `cmd1 && cmd2` **(and)**

  - Execution of commands stops after first failure

- `cmd1 ` `` `cmd2` ``

  - Use output of cmd2 as argument for cmd1

  - `mkdir ` `` `whoami` ``

  - `echo ` `` `date` ``

Next Step: Shell Scripts

- Series of individual commands combined into one executable file form a shell script

- Shell is an interpreter for a programming language of the same name

  - Variables

  - Some prog. constructs: conditional, loops, ...

  - Integer arithmetic

  - etc.

Writing a Script

- Make the first line exactly: `#!/bin/bash`

  - Indicates the command interpreter to be used

  - You need it as soon as you start using any bash-specific constructs

- Type your other commands

- Example: file `trivial.sh` contains two lines

```
#!/bin/bash
echo "Hello world"
```

Executing a Script (3 methods)

- **Start a new shell, execute within that shell**

```
chmod u+x my_script.sh
./my_script.sh
```

- **Start a new shell, execute within that shell**

```
bash my_script.sh
```

- **Execute within current shell**

```
source my_script.sh
```

  - All variables defined in my_script.sh now defined in the invoking shell (see `variable.sh`)

Example

- File trivial.sh contains two lines

```
#!/bin/bash
echo "Hello world"
```

- Now to execute the script

```
> chmod u+x trivial.sh
> ./trivial.sh
```

- Note that we used "./trivial.sh" instead of "trivial.sh" to tell the shell to look in the current directory for trivial.sh

- Instead, we could also have modified our PATH environment variable to include "." (we will do that later)

Writing to stdout or stderr

- By default, output goes to stdout

```
#!/bin/bash

echo "Hello world"
```

- Can also send it to sderr

```
#!/bin/bash

echo "Hello world" >&2
```

Shell Variables

- Assignment using equals sign without spaces

  - `i=42`

  - `q="What is the answer"`

- Preface a variable by a dollar sign ($) to reference its value

  - `echo $q $i`

  - `a="The answer is $i"`

- Optionally, enclose in braces

  - `a2="The answers are ${i}s"`

Example 2

```
> chmod u+x variable.sh

> ./variable.sh

Hello World

Value of MYVAR is 3


> echo $MYVAR
```
                    // nothing is output

Example 2 (b)

```
> source variable.sh

Hello World

Value of MYVAR is 3


> echo $MYVAR

3                    // value 3 is output
```

More about Variables

- By default, variables only seen within the shell itself

  - Can delete a variable with `unset`

  - Check what variables "are set": `set`

- To pass variables to other programs invoked within the shell, use the `export` builtin

  - Exported variable becomes environment variable

  - Examples: `inner.sh` and `outer.sh`

- Several built-in environment variables

  - Example: PATH and HOME

  - Affect shell operation (can you remember how?)

Executing a Script Again

- Start a new shell, execute within that shell:

```
./my_script.sh
bash my_script.sh
```

- Execute within current shell

```
source my_script.sh
```

  - All variables defined in `my_script.sh` now defined in the invoking shell

- Example: try the following

```
./outer.sh; echo $MY_VAR
source outer.sh; echo $MY_VAR
```

Accessing Arguments

- $i is the value of the ith argument

- $0 is the name of the program

- $# is the total number of arguments

- Testing the number of arguments received

```
if [ $# -lt 1 ]
then
...
fi
```

More About Conditions

- `test` command, with `[` as special alias

  - Must put **spaces** around `[` and `]`

  - String tests (limited): `[ aabb = aabb ]`

  - Numeric tests: `[ 1 -lt 5]`

  - File tests (very common): `[ -e my-file ]`

  - Logic with `-a` or `-o`

    - e.g., `[ -f $1 -o -d $1 ]`

  - Logic with `&&` or `||`

    - e.g., `[ -f $1 ] || [ -d $1 ]`

- More info: Linux Pocket Guide (pp 168-171)

Summary

- **What we covered today**

  - I/O redirection, pipes, combining commands

  - Introduction to writing scripts

    - Arguments, variables, printing, manipulating files

  - Emacs

- Content of lectures 1 through 3 is enough to complete first assignment

- You have all the information. Assignment 1 helps you practice and review

Readings

- Class website: pointer to online Emacs manual is in the "Resources" section

- Section from the Linux Pocket Guide

  - Programming with Shell Scripts (pages 166-178)

  - Selected bash features (pages 21-29)