

Some Useful Linux/UNIX Things

ICS412 - Fall 2009 Operating Systems

Henri Casanova (henric@hawaii.edu)

What is This “Lecture” About?

- Some people in the class have voiced a desire to have some type of “UNIX refresher”
- So here we go...
 - We could talk about all this for days, but I’ll let you discover things on your own
 - I won’t talk about text editors
 - vim, emacs, etc.
- If you’re familiar with a UNIX/Linux environment, this is going to be boring
 - If you have the book, do some reading
 - If you have a computer, do the programming assignment

Basics

- You’ll be using the Shell
 - Either by SSH-ing into a server
 - Or by logging in to your own Linux (Virtual) box
- There are many kinds of Shell
- The most standard one is probably: bash
 - We’re going to assume bash from now on
- To find out which shell your default is:
 - `echo $SHELL`
 - I use *this font* to denote commands you can type
- SHELL is an environment variables
- There are many other environment variables:
 - `printenv`
 - `echo $HOME`
 - `echo $USER`
- Sometimes you’ll have to set/modify environment variables
- Setting a new environment variable (or overwriting another one):
 - `export NEWTHING= "a:b/c"`
- Adding to a new environment variable:
 - `export NEWTHING= "$NEWTHING hello"`

Changing the Shell

- If you log in to a machine, and the Shell isn’t the one you like, you can always just type, e.g., `bash`
 - The `chsh /bin/bash` command will change your default Shell to bash forever
 - Note that it needs the full path to the bash executable
 - If you don’t pass it a valid path for bash, you’re in trouble
- Finding the path to a command:
 - `which ...`
 - e.g., `which ls`
 - e.g., `which gcc`
- What’s in your path?
 - `echo $PATH`
 - A super important environment variable
- Adding to your path?
 - `export PATH=$PATH:/some/new/directory/for/binaries`

Customizing your Shell

- Default Shell behavior is stored in a file at the root of your directory called `.xxxxrc`, where `xxxx` is the name of your Shell
 - e.g., `.bashrc`
- In that file you can:
 - Create aliases
 - Set environment variables
 - And do a bunch of other things we won’t talk about
- There is an art to `.bashrc` files
 - Changing the prompt is always amusing
 - The Web is full of sample `.bashrc` files, some simple, some less simple
- Let’s look at the basic two things above

Aliases and Env Variables

- In your `.bashrc` file, anywhere, you can have a line like:
 - `alias foo='blah'`
 - From now on, each time you type the `foo` command, the Shell will replace it by the `blah` command
- Highly recommended aliases
 - `alias rm='rm -i'`
 - `alias mv='mv -i'`
 - `alias cp='cp -i'`
- In your `.bashrc` file you can also set environment variables:
 - `export FOO=BLAH`
 - Very useful for the PATH variables
 - `export PATH=$PATH:/home/casanova/bin`
 - Don’t forget to just *add* to the old path, which comes with good default
 - Doing `export PATH=foo` will not be good as your Shell won’t be able to run any commands

Customizing the Shell

- Once you've modified your `.bashrc` file, you need to "reset" the Shell
 - you can log out and back in
 - you can do `source $HOME/.bashrc`

Wildcards

- `pwd`
 - Prints out the current directory
- `ls *.c`
 - Shows the list of all files named `xxxx.c` in the current directory
- `ls -l *.h *.c */*.g`
 - Shows a detailed list of all `xxx.h` and `xxx.c` files in the current directory and all `xxx.g` files in any q-level-deep subdirectory
- `ls -l dc??d*.c`
 - Shows all files in the current directory whose names start with "dc", then 2 arbitrary characters, then "d", and then an arbitrary number (possibly 0) of arbitrary variables

make

- The "make" tool and Makefiles are for compiling/linking software easily
 - automatic
 - customizable
 - recompiles only when necessary if the Makefile is written correctly
- Makefiles can be very simple or very complicated
- For this class you'll need to know the basics
 - And I'll provide a few samples

Commands

- Every command, system program, or API call has a "man page"
 - `man xxxx`
- Commands take arguments, and/or input from stdin, and produce output on stdout
- Commands you know, but that may have tons of cool options you don't know about
 - `ls`, `cp`, `mv`, `rm`, `mkdir`, ...
- Reading man pages is a very worthwhile activity
 - You don't want to be told by your senior co-worker "go read the man page"
- Some man pages are very instructive
 - `man` is a command, and you can do `man man`
- A few key "things":
 - wildcards, `gcc`, `make`, `pwd`, `cat`, `grep`, `|`, `less`, `wc`, `jobs` (`&`, `^Z`, `kill`, `fg`)
- I am just going to go through a bunch of "random" examples

gcc

- `gcc foo.c -o foo`
 - `foo.c` must contain a `main()`
- `gcc -c foo.c -o foo.o`
 - just generate the object file
- `gcc foo.o faa.o -o foo`
 - link together two object files into an executable
- `gcc -g foo.c -o foo`
 - creates an executable with debugging information

Makefile Example

```
all: myprogram

CFLAGS=-pg
CC=gcc
LD=gcc

myprogram: myprogram.o
    $(LD) myprogram.o -o myprogram
myprogram.o: myprogram.c
    $(CC) -c $(CFLAGS) myprogram.c -o myprogram.o

clean:
    /bin/rm -f *.o myprogram
```

Makefile Example: variables

```
all: myprogram

CFLAGS=-pg
CC=gcc
LD=gcc

myprogram: myprogram.o
    $(LD) myprogram.o -o myprogram
myprogram.o: myprogram.c
    $(CC) -c $(CFLAGS) myprogram.c -o myprogram.o

clean:
    /bin/rm -f *.o myprogram
```

Makefile Example: targets

```
all: myprogram
CFLAGS=-pg
CC=gcc
LD=gcc
myprogram: myprogram.o
    $(LD) myprogram.o -o myprogram
myprogram.o: myprogram.c
    $(CC) -c $(CFLAGS) myprogram.c -o myprogram.o
clean:
    /bin/rm -f *.o myprogram
install:
    cp -f ./myprogram /usr/local/bin/
```

default target (all: myprogram)

other targets (myprogram, myprogram.o, clean, install)

% make myprogram
 % make clean
 % make
 defaults to the first target

Makefile Example: dependencies

```
all: myprogram
CFLAGS=-pg
CC=gcc
LD=gcc
myprogram: myprogram.o
    $(LD) myprogram.o -o myprogram
myprogram.o: myprogram.c
    $(CC) -c $(CFLAGS) myprogram.c -o myprogram.o
clean:
    /bin/rm -f *.o myprogram
install:
    cp -f ./myprogram /usr/local/bin/
```

dependencies (arrows pointing from targets to their prerequisites)

Makefile Example: actions

```
all: myprogram
CFLAGS=-pg
CC=gcc
LD=gcc
myprogram: myprogram.o
    $(LD) myprogram.o -o myprogram
myprogram.o: myprogram.c
    $(CC) -c $(CFLAGS) myprogram.c -o myprogram.o
clean:
    /bin/rm -f *.o myprogram
install:
    cp -f ./myprogram /usr/local/bin/
```

actions (arrows pointing from targets to their commands)

WARNING: TAB character before an action!

Makefile Example

- To “make” a *target*, first “make” its *dependencies*, and then do the *actions*
 - If a file with the same name as the target exists, then do nothing!
- Example:


```
myprogram: myprogram.o
    $(LD) myprogram.o -o myprogram
```

 - To make target “myprogram” I need the myprogram.o object file. If I have that file, then I run the command-line “gcc myprogram.o -o myprogram”
 - If I don’t have the object file, I look for a target called “myprogram.o”
 - If I don’t find such a target, then there is an error

Makefile

- The “clean” target
 - Convention
 - “make clean” should remove all products of compilation
 - useful to send the code to somebody
- The “install” target
 - Convention in GNU software
 - “make install” will put all the right pieces in the right place in your system (provided you are superuser)

Using make

- More information on make
 - http://vertigo.hsrl.rutgers.edu/ug/make_help.html
- Typical sequence of actions when installing GNU software
 - gunzip foo.tar.gz
 - tar -xf foo.tar
 - cd foo
 - ./configure
 - make
- You will have to write Makefiles for your programming assignments

cat and grep

- The cat command takes as argument a file and sends it to stdout (you can see it in the Shell terminal)
 - cat file.c
- grep finds a string in a file or in a set of files and prints the corresponding lines to stdout
 - grep main file.c
 - grep foo *.c */*.c
 - Wildcards: * and .

|, less, wc

- | is used to “pipe” commands together
 - The standard output on one go to the standard input of the next one
- less: sends a file to stdout but wait for user input to display more than the window size
 - e.g., cat file.c | less
- wc: counts lines, words, and characters in a file (-l for counting lines)
 - e.g., cat *.c | grep pthread_create | wc -l
 - counts the number of lines of code that contain “pthread_create”
 - e.g., ls | grep “a.c” | wc -l
 - counts the number of files that contain a.c
 - The “\” is used to “escape” the “.” character, which is special (grep uses it as a wildcard)

Job management

- You can always start a command “in the background” with the & symbol
 - ls -R | wc -l &
- You get control right away
- jobs is used to look at running “jobs”
- jobs can be accessed as %1, %2, ...
- fg %2 brings job #2 to the foreground
- If a job is already running, hitting ^Z suspends the job and gives it a job id
- bg %4 resumes suspended job in the background
- kill %7 kills jobs #7
 - kill -9 %7 is more violent

That's it for Now

- The Shell is much more powerful than many people think and can do a lot for you
- Obviously we've only scratched the surface
- Bash scripts are real programs
- Being a Shell expert will impress your co-workers
- For this course you need to not waste time on Shell issues
 - Ask questions in class right away
- Knowing a scripting language (Perl, Python, etc.) is a good idea for your future
 - Could be useful for assignment #1 to avoid a bunch of by-hand work
 - Most people these days don't really learn much Shell programming and do everything in better scripting languages for rapid development
- Google is your friend for resolving Shell issues!