Jaiswal Lab Linux and Programming Workshop

Justin Elser

8/9/10

Linux

- Linux was developed originally by one Icelandic student, Linus Torvalds in 1991

- Linux is open source

- Technically, Linux is just the kernel

- The rest is other open source packages developed to work with the kernel

- The compilation of these packages are packaged in “distributions”


Home > Desktop
dosbox-pykde
Handbrake
k9copy
Mozilla Firefox
Mozilla Thunderbird
My Computer
Office
Starcraft+-+Brood+War....
TVUPlayer
Unison
VirtualBox
VLC
Gecko
Taskbar
Layers, Channels, Paths
GNU Image Manipulation Prog
Untitled 1 - OpenOffice.org In
100%
03:51 pm

Common programs

- Jedit
- OpenOffice
- Firefox
- Eclipse
- Konsole (shell, terminal...)

Shell commands in Linux

ls – list all commands in a directory (list)

switches:
-l long list format
-a show all files
-t sort by time modified

switches can be combined (ls -lrt)
aliases (ll, la)

Shell file commands in Linux

mkdir – make a directory
cd – change directory
df – disk space file usage (shows mount points)
rm – remove a file
rmdir – remove a directory
mv – move or rename a file
cp – copy a file
pwd – print working directory
more/less – pagers (view a file one page at a time
man – show the manual page for a command
use 'man -k *term'* to look for commands that have a specific *term* in them

Shell environment variables

$HOME, ~

$PATH

$CLASSPATH

$SHELL

..

./

Remote commands

rsh – remote shell

telnet – remote login

ftp – file transfer protocol

ssh – secure shell

sftp – secure file transfer protocol

scp – secure copy (machine to machine)

Viewing a file

tail – show the last 10 lines of a file

head – show the first 10 lines of a file

cat – show the entire file

Redirects:
> - create a file (head file.txt > short_file.txt)
((will give error if short_file.txt exists))

>> - append to a file (head file.txt >> short_file.txt)

matching text in a file

grep – print matching lines in a file (or input)

$ grep apple fruitlist.txt

Works with regular expressions (more on regex later)
$ egrep [Aa]pple fruitlist.txt

doesn't have to match the entire line
$ grep pple fruitlist.txt

stringing commands together

| - pipe command

$ cat file.txt | grep text > output.txt

You can combine as many commands together you wish. The output from one becomes the input to the next.

The above is sometimes called the “useless use of cat”, but works as an illustration.

editing text on the fly

sed – stream editor

Has several ways to do things, each with a different syntax. For us, we mostly will just use the following:
sed 's/old_text/new_text/g'

$ cat fruitlist.txt | sed 's/apple/banana/g'

breakdown:
's – search, /apple – term to search for, /banana – term to replace with, /g - global

picking out parts of a line

awk – programming language used for dealing with text

cat sample.txt | awk '{print $2}'
this will print out the second field of each line of the sample.txt file

Use the -F switch to change field seperators (-F “,” or -F “\t”)

sorting and finding unique lines

sort and uniq

cat fruit.txt | sort | uniq

Will sort the fruit.txt file alphabetically and then only show unique results.

Note, be careful when sorting numbers, because by default it will sort by the first digit:

cat numbers.txt | sort
1000
20
555

other useful commands to know

history – shows the last commands run in the shell

ps – show running processes

top – show all running processes + more info

wc – count the number of lines, bytes, etc...

cgrb_mount – mounts the shared lab space

Exercises:

Please do as many of the problems on the handout as you can. Write down your solutions.

Perl

high-level, interpreted programming language

Created by Larry Wall in 1987

Available for Linux, Mac, and Windows (ActivePerl)

Basic Perl code

The 1st 3 lines of code should always be:

```
#!/usr/bin/perl
use strict;
use warnings;
```

Comments:

Start with a # for single line comments, does not support multiline comments

```
#!/usr/bin/perl
use strict;
use warnings;
# This program is a basic perl program
```

Data types:

Perl has 3 data types:

scalar – pretty much anything you want to store, such as numbers, strings, binary, booleans...

array – sequential list of scalars

hash – associated array

Scalars:

Declaring a scalar:

```
my $string = “text”;
my $num = 24;
my $bin = 0b0; # probably will never use this
my $bool = undef;
my $func = 2-3+7; # $func = 6
my $new_string = $string;
```

Combining strings:

```
my $string_1 = “Hello”;
my $string_2 = “world”;
my $string_comb = $string_1 . “ “ . $string_2;
```

Printing:

```
print “$string_comb\t\n”;
```

Arrays:

Arrays are lists of scalars that are indexed sequentially starting from 0

```
my @array = (3,1,4,1,5,9);
print “$array[3]”; #result will be “1”
```

array sizes are dynamic, add new elements with push:

```
push(@array,2);
push(@array,6,5,3);
```

remove with pop:

```
my $last_digit = pop(@array);
```

shift and unshift do the same as push and pop, but to the beginning of the array rather than the end

Hashes:

Hashes are associative arrays. This means that rather then being numerically indexed, you control the index.

```
my %hash; # index is person, value is age
$hash{“justin”} = 33;
$hash{“preece”} = 35;
```

The advantage to hashes is that you can look up specific info rather than having to loop through an entire array searching:

```
my $justin_age = $hash{“justin”};
print “$justin_age\n”; #result is 33
print “justin is\t$hash{“justin”}; # justin is  33
```

Use arrays if going to loop through every value, hashes if you want to be able to look for specific values.

File I/O:

Create filehandles:

```
open(in_file, “input.txt”) || die “Error: file could not be opened\n”;
open(out_file, “>output.txt”) #writable
```

Reading from file:

```
while(<in_file>) {
    my $line = $_;
    chomp $line; #remove newline characters
    print out_file “$line\n”;
}
```

Logic:

Statements that are either true or false.

Computer definition of true and false:

0 #converts to “0”, so false
1-1 #computes to 0, then converts to “0”, so false
1 #true
“” # empty string, so false
“00”
undef

Comparisons:

2>3 #false
3.0 == 3 #true in perl, not in other languages
“apple” eq “Apple” #false
“apple” gt “banana” #false, use to alphabetize

others:
!=, <=, ne, le

Loops:

Loops are logic conditional statements that execute a block of code. We already saw a while loop in the file I/O section. Other loops include:

```
for (my $counter = initial; logic conditional; counter step) {code}

foreach my $element (@array) {code} #will run for every element

if(logic condition) {code
}else{other code
}

if(logic condition) {code
}elsif{other code
}else{yet more code
}
```

Scope:

Perl is called “lexically scoped”. This means that variables declared inside a block of code are unavailable outside that block:

```
my $outer_sum;
my @array = (1,1,2,3,5,8,13);
my $array_size = @array;
for (my $i = 0; $i<$array_size; $i++) {
    my $digit = $array[$i];
    $outer_sum +=$digit;
}
print “$outer_sum\n”; # 33
print “$digit\n”; #error
```

Regular Expressions:

Regular expressions are patterns to be matched against a string.

Examples:

/[abcde]/ # matches lowercase a, b, c, d, e
/[aeiouAEIOU]/ # matches upper or lowercase vowels
[0123456789] is the same as [0-9]
[a-z0-9] # matches lowercase letters and any digit
[^0-9] # matches non-digits

Constructs:

\d # digit, equivalent to [0-9]
\w # word, equivalent to [a-zA-Z0-9_]
\s # space character, [\r\t\n\f]
\D # not a digit, [^0-9]

Anchors:

/^apple/ # matches apple but not cranapple
/apple$/ # matches apple (at the end of a line), but not apples

RegEx, cont.:

```
my $string = “hello world”;
$string =~ /^he/; #true
if ($string =~ /^he/) {
      do some code}
```

Substitutions:

```
$string =~ s/hello/Hello/;
$string =~ s/world/World/;
```

Splitting strings:

```
my $line = “apple  banana      cucumber  danish”;
my @fields = split(“\t”, $line);
my ($one,$two,$three) = split(“:”, $line);
```

Transliteration:

```
my $string = “apple”;
$string =~ tr/ap/pa/;
print “$string\n”; #paale
```

Exercises:

Please do as many of the problems on the handout as you can. Keep your program files.

SVN and SGE:

SVN (subversion) is a software version control system. We use it in our lab to keep track of versions of software that we write. It is useful because it allows you to see differences in your programs as they are modified, and acts as a backup.

SGE is the “Sun Grid Engine”. It is the software that controls how the cluster is managed. For end-users, it is how you interact with the cluster, submitting jobs and running long running programs.

SVN:

A central repository hosts the files

Anyone can “checkout” files from the repository

Only users that have been granted access can submit or change items in the repository

For our lab:

All programs should be submitted to the repository (well, longer than a few lines)

Data is not to be hosted in the repository, that is what the lab share is for

Some of the files that I have done in there are to be “standard” libraries for us to use. For example, I have written code to assist with accessing the database or finding the species given certain info or the gene from headers from our collection of sequence files

SVN commands:

checkout – use to checkout code from the repository:
svn co http://palea.cgrb.oregonstate.edu/svn/jaiswallab (--username=elserj)

update – update to the latest version of the repository

commit – write the changes you have made to the repository

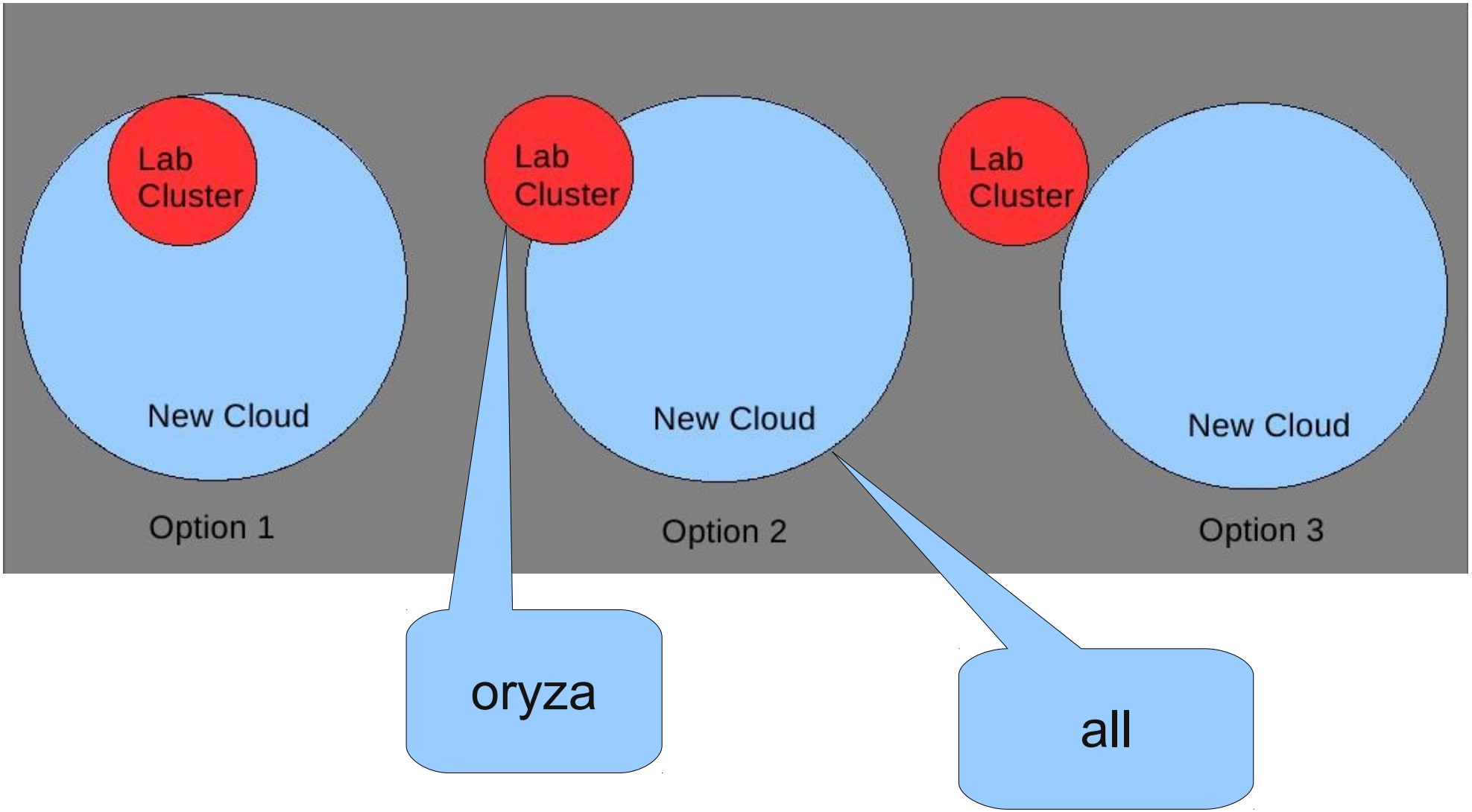
add – add a new file (or directory) to the repo

remove – remove a file from the repo. Note that older versions will still be available

info – give information about the file from the repo (version, last changed revision, etc..)

merge – if 2 (or more) people have modified the same file such that there are conflicts, you can merge the files together.

CGRB Cluster:



SGE:

Sun Grid Engine – cluster management software

allocates resources to users via “q” commands

qrsh – remote shell login to a node;
qrsh -q oryza.q

qsub – submit a job to the queue

qstat – show running jobs. By default, only shows yours. Use qstat -u '*' to show all users

qdel *jobid* – stop a running (or queued) job

Writing your own qsub script:

qsub cannot accept (binary) programs directly. You must create a submit script to do it.

For serial jobs (single processor), this is fairly trivial to do.

#!/bin/bash
/path/to/executable (or you can use any command in a shell script)

You can then submit the script via qsub:
qsub submit_script
qsub -q oryza submit_script

check the status of your job:

```
[elserj@waterman ~]$ qstat
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
  63968 0.00000 justin     elserj       qw    08/12/2010 10:38:45                                    1
```

SGE_Batch:

Script written by CGRB to help with writing submit scripts for SGE

Available on waterman.cgrb.oregonstate.edu

Launch interactively:
SGE_Batch
Use the options to set the command (program) to run and the output directory. Optionally, you can have it email you when job is done, or to use a machine with a minimum memory amount.

Launch directly:
SGE_Batch -c 'echo $HOSTNAME' -o sge_output
Note: sge_output must be relative to current directory and be on the same level

Exercises:

Do as many as you can.