CSE 303
Lecture 6

more Unix commands;

bash scripting continued

read *Linux Pocket Guide* pp. 66-68, 82-88, 166-178

slides created by Marty Stepp

http://www.cs.washington.edu/303/

Remote connections

command	description
`ssh`	open a shell on a remote server
`sftp`	open a connection to transfer files to/from a server
`scp`	copy files to/from a server, then disconnect

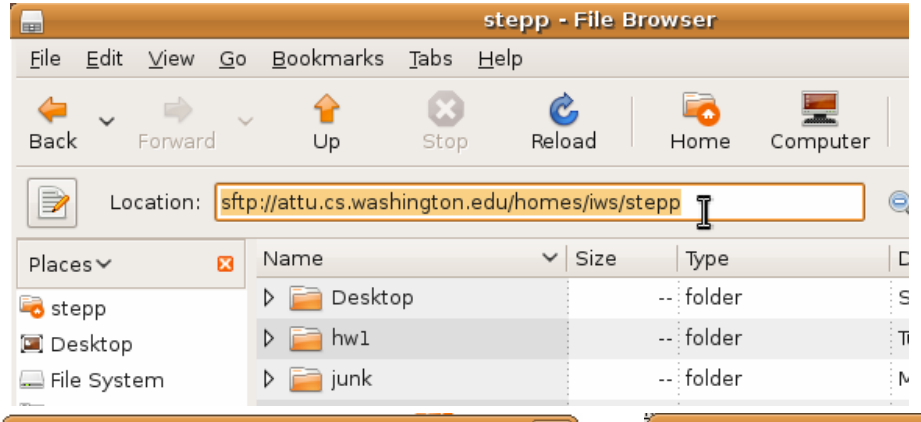
- `sftp` ***`servername`***
  - once connected, can use `cd`, `ls`, `PUT` ***`filename`***, `GET` ***`filename`***

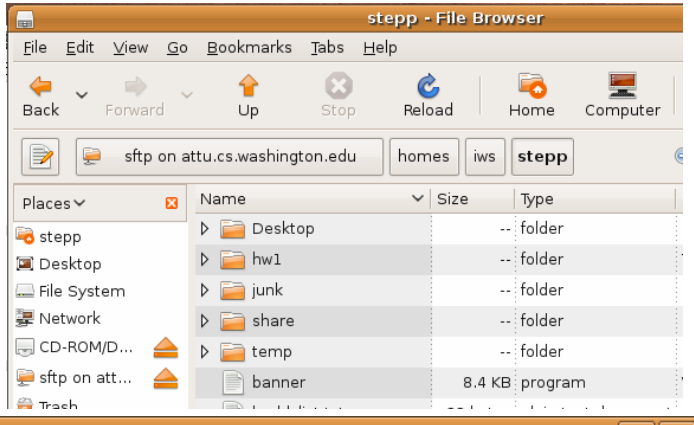
- `scp` ***`filename(s) user@server:/path/file`***
  - Examples:

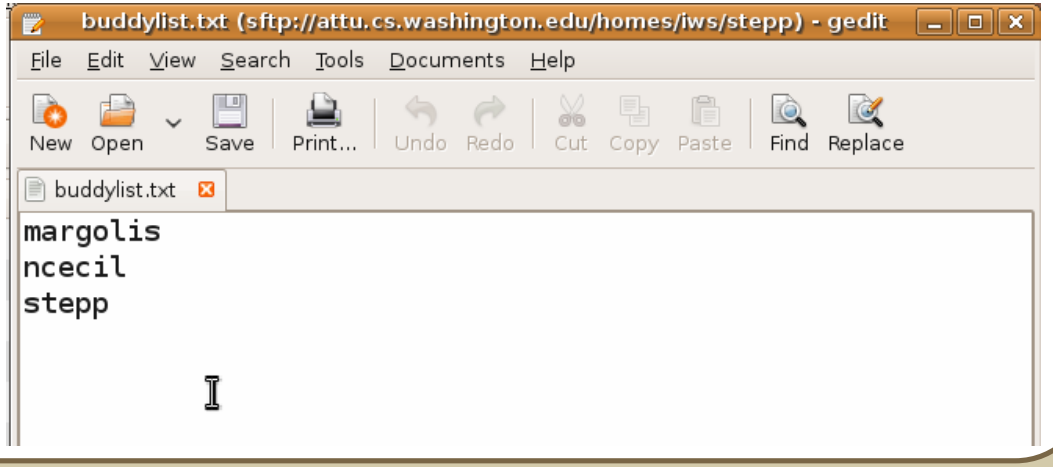
```
scp * stepp@attu:/homes/iws/stepp/hw1
scp stepp@attu:/homes/iws/stepp/* .
```

Remote editing

- Gnome's file browser and `gedit` text editor are capable of opening files on a remote server and editing them from your computer
  - press Ctrl-L to type in a network location to open



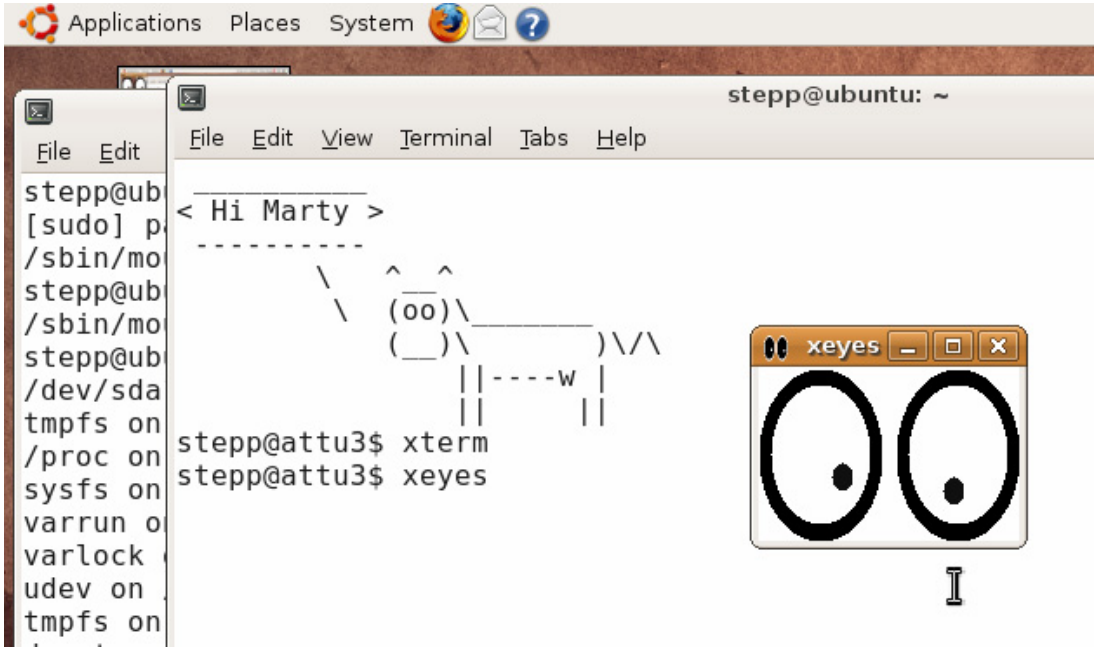




Remote X display

- normally, you cannot run graphical programs on a remote server
- however, if you connect your SSH with the -Y parameter, you can!
  - the X-Windows protocol is capable of displaying programs remotely

```
ssh -Y attu.cs.washington.edu
```



Compressed files

command	description
`zip, unzip`	create or extract .zip compressed archives
`tar`	create or extract .tar archives (combine multiple files)
`gzip, gunzip`	GNU free compression programs (single-file)
`bzip2`	slower, optimized compression program (single-file)

- many Linux programs are distributed as `.tar.gz` archives
  - first, multiple files are grouped into a `.tar` file (not compressed)
  - next, the `.tar` is compressed via `gzip` into a `.tar.gz` or `.tgz`

- to decompress a `.tar.gz` archive:

```
$ tar -xzf filename.tar.gz
```

Comparing files

command	description
`diff`	outputs differences between two text files

- Example:

```
$ diff file1.txt file2.txt
1c1
< Hello!
---
> Hi!
5d4
< Thanks!
7a7
> So long.
```

file1.txt	file2.txt
Hello! How are you? I am just fine. **Thanks!** Well, goodbye.	**Hi!** How are you? I am just fine. Well, goodbye. **So long.**

Searching for files

command	description
`find`	searches for files in a given directory tree (recursively processes subdirectories)

`find` ***`path`*** `-name` ***`pattern`***

- Examples:

```
$ find . -name *              (find all files)
$ find foo/ -name *.txt       (find .txt files in foo/ )
```

- Often used with `xargs` to apply an operation to each found file:

```
$ find . -name *.sh | xargs chmod +x
```
(make all `.sh` scripts executable)

Searching in files

command	description
grep	searches for patterns of text within a file

- Character-set patterns:
  - [abcd] - lines that have an a, b, c, or d
  - [abcd]efg - lines that have an (a, b, c, or d) followed by efg
  - [abcd]* - lines that contain strings of as, bs, cs, and/or ds

- Example:

```
$ grep "CSE 14[23]" homework/*
```

More Shell Scripting

if/else

```
if [ test ]; then                # basic if
    commands
fi

if [ test ]; then                # if / else if / else
    commands1
elif [ test ]; then
    commands2
else
    commands3
fi
```

- there ***MUST*** be a space between `if` and `[` and between `[` and ***`test`***
  - `[` is actually a shell command, not just a character
  - also be careful to include the comma between `]` and `then`

Testing commands

shell command	description
=, !=, <, >	compares two string variables
-n, -z	tests whether a string is or is not empty (null)
-lt, -le, -eq, -gt, -ge, -ne	compares numbers; equivalent to Java's <, <=, ==, >, >=, !=
-e, -d	tests whether a given file or directory exists
-r, -w	tests whether a file exists and is read/writable

```
if [ $USER = "stepp" ]; then
    echo "Hello there, beautiful!"
fi

LOGINS=`w | wc -l`
if [ $LOGINS -gt 10 ]; then
    echo "attu is very busy right now!"
fi
```

More if testing

shell command	description
`if [ expr1 -a expr2 ]; then ...`	and
`if [ expr1 -o expr2 ]; then ...`	or
`if [ ! expr ]; then ...`	not

```
# alert user if running >= 10 processes when
# attu is busy (>= 5 users logged in)
LOGINS=`w | wc -l`
PROCESSES=`ps -u $USER | wc -l`
if [ $LOGINS -gt 5 -a $PROCESSES -gt 10 ]; then
    echo "Quit hogging the server!"
fi
```

Command-line arguments

variable	description
`$0`	name of this script
`$1, $2, $3, ...`	command-line arguments
`$#`	number of arguments
`$@`	array of all arguments

```
if [ "$1" = "-r" ]; then
    echo "Running in special reverse format."
fi

if [ $# -lt 2 ]; then
    echo "Usage: $0 source destination"
    exit 1      # exit the script, error code 1
fi
```

Exercise

- Write a program that computes the user's body mass index (BMI) to the nearest integer, as well as the user's weight class:

$$BMI = \frac{weight}{height^2} \times 703$$

BMI	Weight class
≤ 18	underweight
18 - 24	normal
25 - 29	overweight
≥ 30	obese

```
$ ./bmi
Usage: ./bmi weight height

$ ./bmi 112 72
Your Body Mass Index (BMI) is 15
Here is a sandwich; please eat.

$ ./bmi 208 67
Your Body Mass Index (BMI) is 32
There is more of you to love.
```

Exercise solution

```
#!/bin/bash
# Body Mass Index (BMI) calculator
if [ $# -lt 2 ]; then
    echo "Usage: $0 weight height"
    exit 1
fi

let BMI="703 * $1 / $2 / $2"
echo "Your Body Mass Index (BMI) is $BMI"
if [ $BMI -le 18 ]; then
    echo "Here is a sandwich; please eat."
elif [ $BMI -le 24 ]; then
    echo "You're in normal weight range."
elif [ $BMI -le 29 ]; then
    echo "You could stand to lose a few."
else
    echo "There is more of you to love."
fi
```

Common errors

- `[: -eq: unary operator expected`
  - you used an undefined variable in an `if` test
- `[: too many arguments`
  - you tried to use a variable with a large, complex value (such as multi-line output from a program) as though it were a simple int or string
- `let: syntax error: operand expected (error token is " ")`
  - you used an undefined variable in a `let` mathematical expression

for and while loops

```
for name in value1 value2 ... valueN; do
    commands
done
```

- the pattern after `in` can be:
  - a hard-coded set of values you write in the script
  - a set of file names produced as output from some command
  - command line arguments: `$@`

```
while [ test ]; do          # not used as often
    commands
done
```

Exercise

- Write a script `createhw.sh` that creates directories named hw1, hw2, ... up to a maximum passed as a command-line argument.

```
$ ./createhw.sh 8
```

  - Copy `criteria.txt` into each assignment *i* as `criteria(2*i).txt`
  - Copy `script.sh` into each, and run it.
    - output: `Script running on hw3 with criteria6.txt ...`
  - If any directory already exists, skip it and print a message such as:

```
You already have a hw3 directory!
```

  - The following command may be helpful:

command	description
`seq`	outputs a sequence of numbers

Exercise solution

```
#!/bin/bash
# Creates directories for a given number of assignments.
if [ $# -lt 1 ]; then
    echo "Usage: $0 MAX"
    exit 1
fi

for num in `seq $1`; do
    if [ -d "hw$num" ]; then
        echo "You already have a hw$num directory!"
    else
        let CNUM="2 * $num"
        mkdir "hw$num"
        cp script.sh "hw$num/"
        cp criteria.txt "hw$num/criteria$CNUM.txt"
        echo "Created hw$num."
        cd "hw$num/"
        bash ./script.sh
        cd ..
    fi
done
```

Arrays

```
name=(element1 element2 ... elementN)

name[index]=value          # set an element
$name                      # get first element
${name[index]}             # get an element
${name[*]}                 # elements sep.by spaces
${#name[*]}                # array's length
```

- arrays don't have a fixed length; they can grow as necessary
- if you go out of bounds, shell will silently give you an empty string
  - you don't need to use arrays in assignments in this course

Functions

```
function name() {          # declaration
    commands
}

name                       # call
```

- functions are called simply by writing their name (no parens)
- parameters can be passed and accessed as $1, $2, etc. (icky)
  - you don't need to use functions in assignments in this course