



# Today's Agenda

---

- Further information regarding Class Project
- Finish up Advanced Shell Concepts
  - I/O Redirection & Pipes
  - Environment Variables
  - File permissions & a security issues
- Version Control Introduction
- Discuss Lab Assignment #1
- Assignment for Thursday

# **CSCI 553: Networking III**

## **Unix Network Programming**

Spring 2007



---

More Shell Basics



# Introduction

---

- The shell is more than just a clumsy way to move around a file system
- It's a component-based programming environment
  - Small tools that each do one job...
  - ...can be connected together to create ad hoc solutions to larger problems
- A good model, even when you're building large GUI or web applications



# Wildcards

---

- Some characters (called *wildcards*) mean special things to the shell
    - \* matches zero or more characters
      - So `ls bio/*.txt` lists all the text files in the bio directory
- ```
$ ls bio/*.txt  
bio/albus.txt bio/ginny.txt bio/harry.txt bio/hermione.txt bio/ron.txt
```
- ? matches any single character
    - So `ls jan-?.?.txt` lists text files whose names start with "jan-" followed by two characters
    - You can probably guess what `ls jan-??.*` does
  - Note
    - The shell expands wildcards, not individual applications
      - `ls` can't tell whether it was invoked as `ls *.txt` or as `ls earth.txt venus.txt`
    - Wildcards only work with filenames, not with command names
      - `ta*` does *not* find the tabulate command

# Redirecting Input and Output

- A running program is called a *process*
- Every process automatically has three connections to the outside world:
  - *Standard input* (stdin): connected to the keyboard
  - *Standard output* (stdout): connected to the screen
  - *Standard error* (stderr): also connected to the screen
    - Used for error messages
- You can tell the shell to connect standard input and standard output to files instead
  - `command < input_file` reads from `input_file` instead of from the keyboard
    - Don't need to use this very often, because most Unix commands let you specify the input file (or files) as command-line arguments
  - `command > output_file` writes to `output_file` instead of to the screen
    - Only "normal" output goes to the file, not error messages
  - `command < input_file > output_file` does both
- Note that redirection takes effect command-by-command, rather than permanently

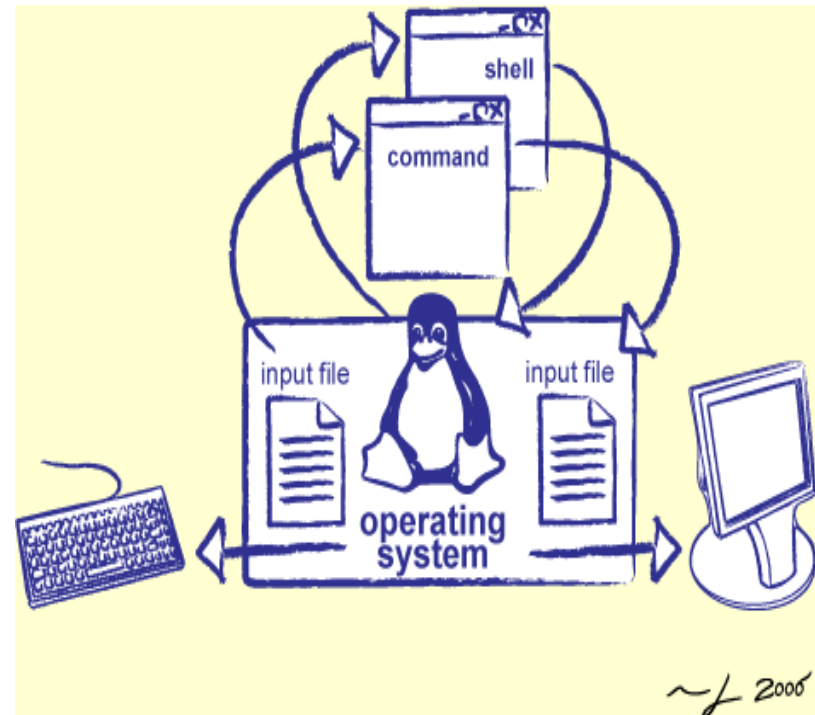


Fig L3.1: Redirecting Standard Input and Output



# Redirection Examples

---

- Save number of words in all text files to words.len:

```
$ cd bio
$ wc *.txt > words.len
```

  - Nothing appears on the screen because output is being sent to the file words.len
  - Check contents using cat

```
$ cat words.len
7 66 468 albus.txt
5 46 311 ginny.txt
5 49 342 harry.txt
5 49 331 hermione.txt
6 54 364 ron.txt
28 264 1816 total
```
- Try typing `cat > junk.txt`
  - No input file specified, so cat reads from the keyboard
  - Output sent to a file
  - The world's dumbest text editor
- When you're done, use `rm junk.txt` to get rid of the file
  - Don't type `rm *` unless you're really, really sure that's what you want to do...
- Don't redirect out to the same file, e.g. `sort words > words`
  - The shell sets up redirection before running the command
  - Redirecting out to an existing file truncates it make it empty
  - `sort` then goes and reads the empty file
  - Contents of words are lost



# Pipes

---

- Suppose you want to use the output of one program as the input of another
  - E.g., use `wc -w *.txt` to count the words in some files, then `sort -n` to sort numerically
- The obvious solution is to send output of first command to a temporary file, then read from that file

```
$ wc -w *.txt > words.tmp
$ sort -n words.tmp
46 ginny.txt
49 harry.txt
49 hermione.txt
54 ron.txt
66 albus.txt
264 total
$ rm words.tmp
```
- The *right* answer is to use a *pipe*
  - Written as "`|`"
  - Tells the operating system to connect the standard output of the first program to the standard input of the second

```
$ wc -w *.txt | sort -n
46 ginny.txt
49 harry.txt
49 hermione.txt
54 ron.txt
66 albus.txt
264 total
```
  - More convenient and less error prone than temporary files.

# Pipes

- Can chain any number of commands together
    - And combine with input and output redirection
- ```
$ grep 'Title' spells.txt | sort | uniq -c | sort -n -r | head -10 > popular_spells.txt
```
- Any program that reads from standard input and writes to standard output can use redirection and pipes
    - Such programs are often called *filters*
    - If your programs work like filters, you (and other people) can combine them with standard tools
    - A combinatorial explosion of goodness

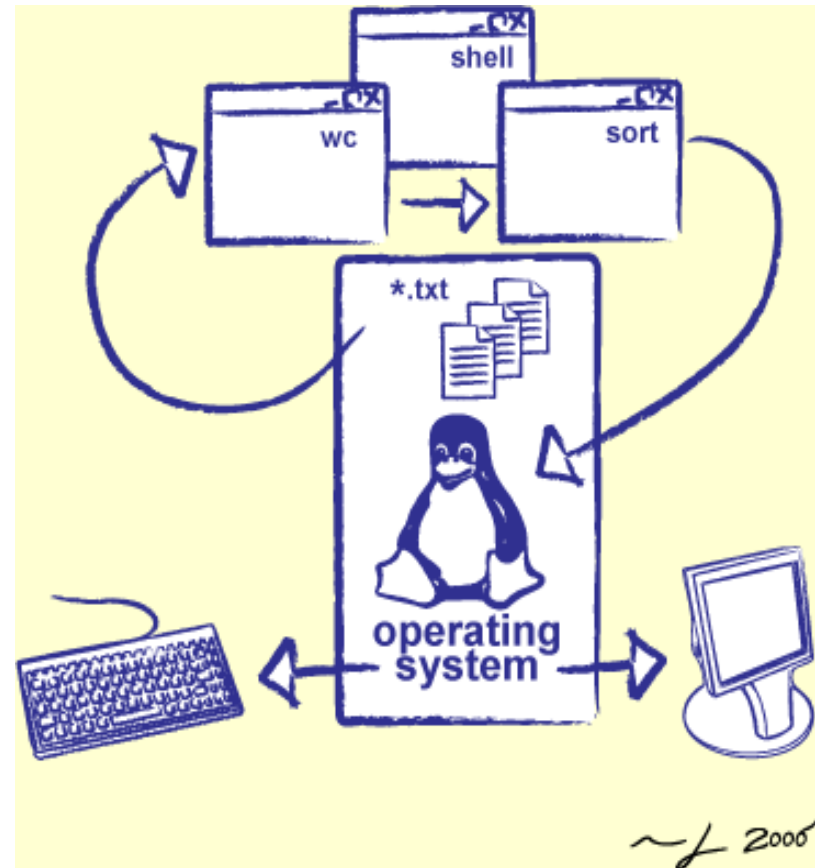


Fig L3.2: Pipes



# Environment Variables

---

- The OS stores some *environment variables* for every process
  - Like variables in a program, each has a name and a value
    - By convention, names are all upper case
    - Values are always strings
- Type set at the command prompt to get a listing:

```
$ set
BASH=/usr/bin/bash
BASH_VERSION='2.05b.0(1)-release'
COLUMNS=120
HISTFILE=/home/.bash_history
HISTFILESIZE=500
HISTSIZE=500
HOME=/home/rweasley
HOSTNAME=hogwarts
HOSTTYPE=i686
LINES=60
NUMBER_OF_PROCESSORS=1
OSTYPE=cygwin
PATH='/usr/local/bin:/usr/bin:/bin:/Python24:/home/rweasley/bin'
PWD=/home/rweasley
SHELL=/bin/bash
UID=1003
USER=rweasley
```



# Environment Variables

---

- Get a variable's value by putting "\$" in front of its name
  - So `ls $HOME` is the same as `ls /home/csci553/rweasley` (if you're Ron Weasley)
  - Use the `echo` command to print out a variable's value
    - `$ echo $HOME`  
`/cygdrive/c/home/rweasley`
    - Question: why must you type `echo $HOME`, and not just `$HOME`?
  - If a variable hasn't been defined, its value is the empty string (rather than an error)

# Important Environment Variables

<b>Name</b>	<b>Typical Value</b>	<b>Notes</b>
EDITOR	/bin/vi	Preferred editor
HOME	/home/rweasley	The current user's home directory
HOSTNAME	"nisl.tamu-commerce.edu"	The computer's name
PATH	"/home/rweasley/bin:/usr/local/bin:/usr/bin:/bin"	Where to look for programs
PWD	/home/rweasley/swc/lec	Present working directory
SHELL	/bin/bash	What shell is being run
TEMP	/tmp	Where to store temporary files
USER	"rweasley"	The current user's ID



# Setting Environment Variables

---

- Different shells have different syntaxes for setting environment variables
- For Bash, use this:  
\$ VILLAIN="Lord Voldemort"
  - Notice that values with spaces in them have to be quoted
- Setting an environment variable only affects that program (i.e., that shell)
  - To see this, set a variable, then run a new shell, and echo the variable's value  
\$ VILLAIN="Lord Voldemort"  
\$ bash  
\$ echo \$VILLAIN  
  
\$ exit  
\$ echo \$VILLAIN

# Setting a Variable Without Exporting It

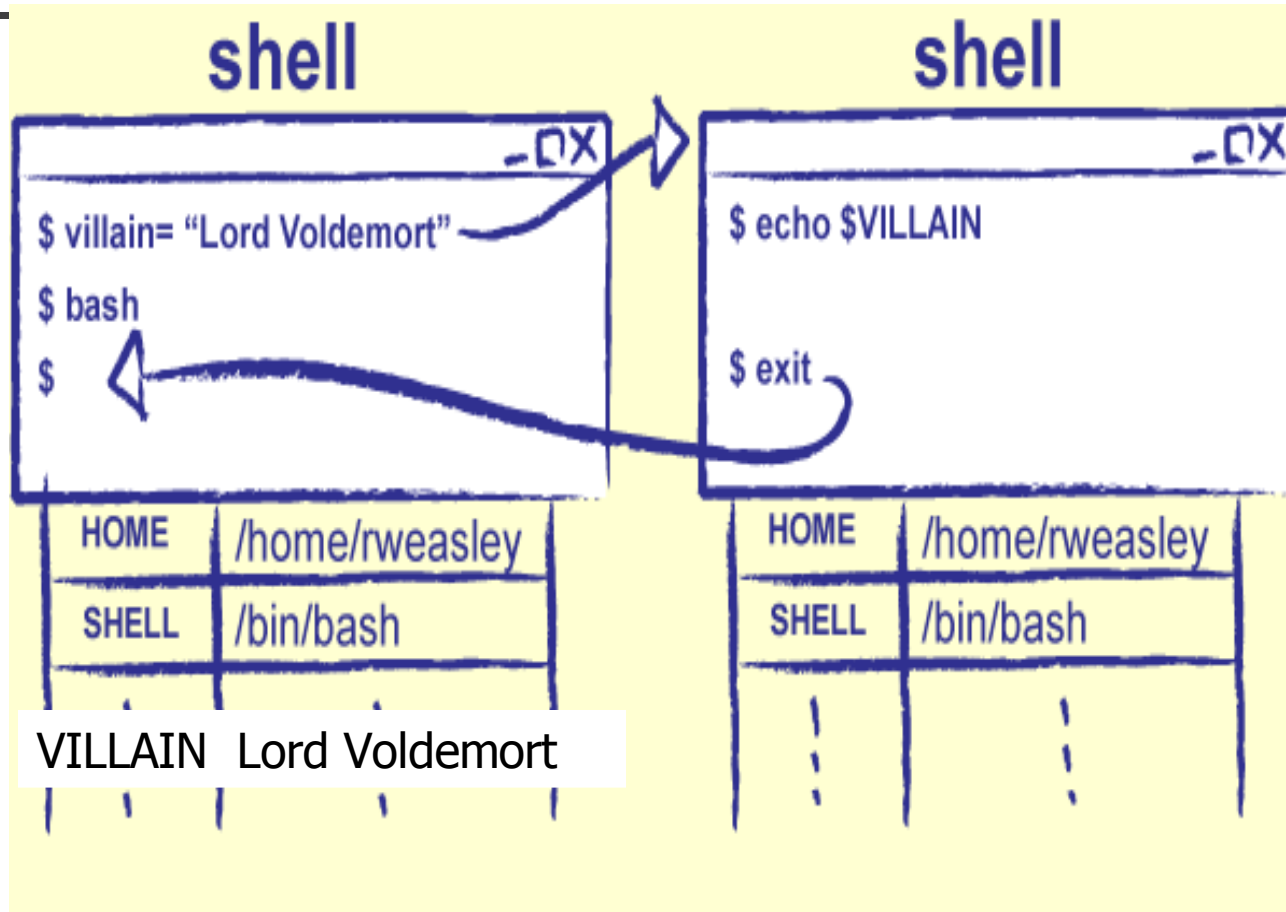


Fig L3.3: Setting a Variable Without Exporting It.



# Setting Environment Variables

---

- If you want programs run from that shell to inherit the value, you must export it:

```
$ VILLAIN="Lord Voldemort"
```

```
$ export VILLAIN
```

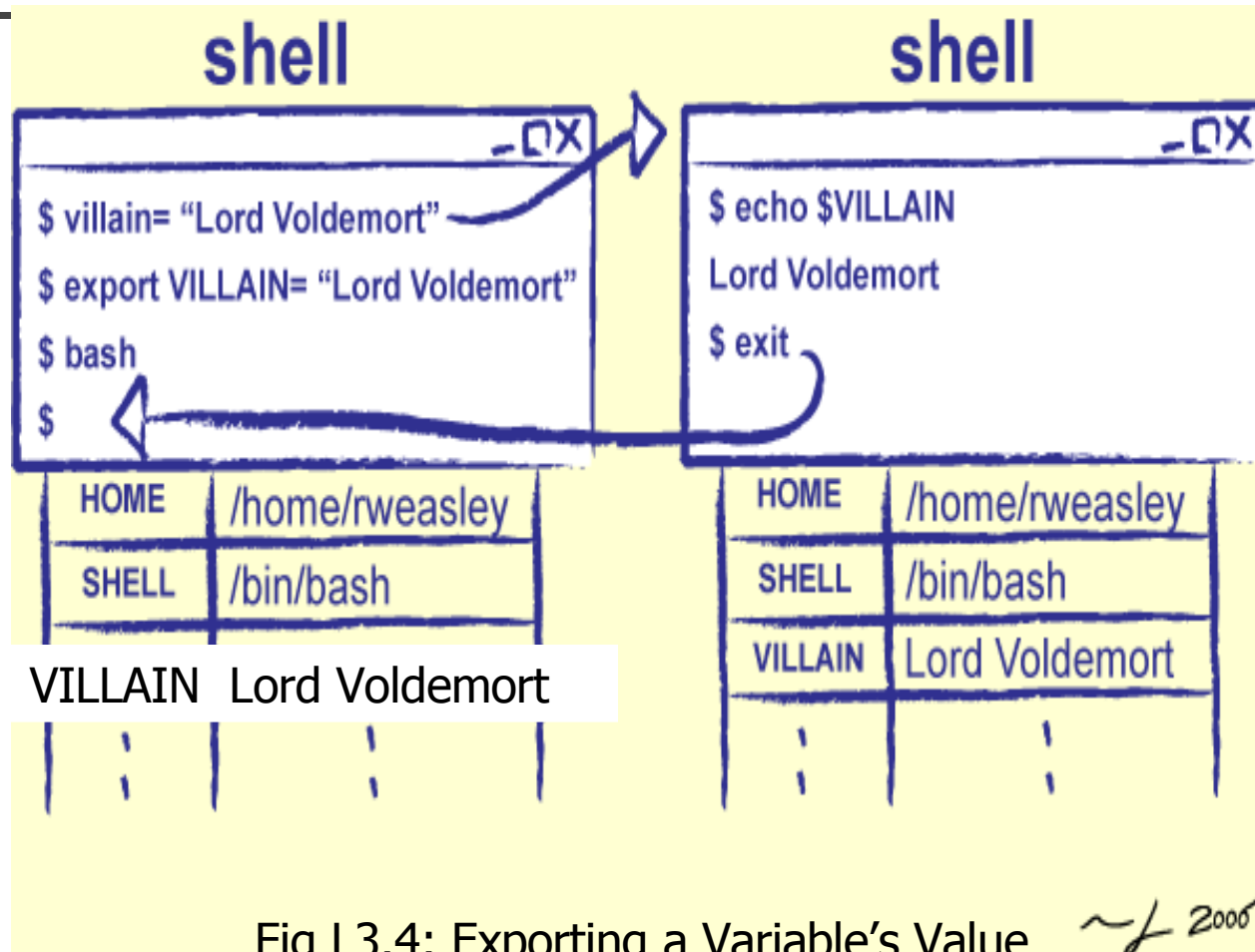
```
$ bash
```

```
$ echo $VILLAIN
```

```
Lord Voldemort
```

```
$ exit
```

# Exporting an Environment Variable





# Setting Environment Variables

---

- Can perform both operations in a single step

```
$ export VILLAIN="Lord Voldemort"
```

```
$ bash
```

```
$ echo $VILLAIN Lord Voldemort
```

```
$ exit
```



# Configuration

---

- To set a variable's value automatically when you log in, edit `~/.bashrc`
  - "`~`" is a shortcut meaning "your home directory"

```
# Add personal tools directory to PATH.  
PATH=$HOME/bin:$PATH  
  
# Personal settings.  
export EDITOR=/local/bin/emacs  
export PRINTER=gryffindor-laserwriter  
  
# Change default behavior of commands.  
alias ls="ls -F"
```
  - Note: `.bashrc` files can become very complex...
- Many applications look for personal configuration files in the user's home directory
  - By convention, their names begin with `."` so that a normal `ls` won't show them
  - Once upon a time, the `"rc"` at the end meant "run commands"



# How the Shell Finds Programs

---

- The PATH environment variables defines the shell's *search path*
- When you run a command like broom, the shell:
  - Splits \$PATH into components to get a list of directories
    - Unix uses ":" as a separator
    - Windows uses ";"
  - Looks for the program in each directory in left-to-right order
  - Runs the first one that it finds
- Example
  - PATH is /home/rweasley/bin:/usr/local/bin:/usr/bin:/bin:/Python24
  - Both /usr/local/bin/broom and /home/rweasley/bin/broom exist
  - /home/rweasley/bin/broom will be run when you type broom at the command prompt
  - Can run the other one by specifying the path, instead of just the command name



# Common Search Path Entries

---

- `/bin, /usr/bin`: core tools like `ls`
  - Note: the word “bin” comes from “binary”, which is geekspeak for “a compiled program”
- `/usr/local/bin`: optional (but common) tools, like the `gcc` C compiler
- `$HOME/bin`: tools you have built for yourself
  - Remember, `$HOME` is your home directory
- It is also common to include `.` (the current working directory) in your path
  - Allows you to run a program in the current directory using `whatever`, instead of `./whatever`

# File Ownership and Permissions



---

- On Unix, every user belongs to one or more groups
  - The groups command will show you which ones you are in
- Every file is owned by a particular user *and* a particular group
  - Can assign read (r), write (w), and execute (x) permissions independently to user, group, and others
  - Read: can look at contents, but not modify them
  - Write: can modify contents
  - Execute: can run the file (e.g., it's a program)
- ls -l shows this information
  - Along with the file's size and a few other things
- Permissions displayed as three rwx triples
  - "Missing" permissions shown by "-"
  - So rw-rw-r-- means:
    - User and group can read and write
    - Everyone else can read, but not write
    - No one can execute



# Directory Permissions

---

- Execute permission means something different for directories
  - Allows you to “go into” a directory, but does *not* mean you can read its contents
- If tools has permission `rwX--X--X`, then:
  - If someone other than the owner does `ls tools`, permission is denied
  - But anyone who wants to can run `tools/pfold`



# Changing Permissions

---

- Change permissions using chmod
  - `chmod u+x broom` allows broom's owner to run it
  - `chmod o-r notes.txt` takes away the world's read permission for notes.txt
- Any set of shell commands can be turned into a program!
  - If it's worth doing again, it's worth automating
- Create a file called nojunk

```
#!/usr/bin/bash  
rm -f *.junk
```

  - Use `man rm` to find out what the `-f` flag does
  - `#!/usr/bin/bash` means "run this using the Bash shell"
    - Any program name can follow the `#!`
    - We'll see some possibilities later
- Change permissions to `rxr-xr-x`
- Run it with `./nojunk`
  - Or if `$HOME/bin` is in your search path, move it there
- Don't call your temporary test programs test
  - There's already `/usr/bin/test`
  - Your `PATH` may cause that program to run instead of yours
  - Confusion results, so use something else, e.g. `./try`



# More Advanced Tools

---

\* Especially these

<b>chmod *</b>	Change file and directory permissions.
<b>du</b>	Print the disk space used by files and directories.
<b>find *</b>	Find files with names that match patterns, that are of a certain age or size, etc.
<b>grep *</b>	Print lines matching a pattern (uses regular expressions)
<b>gunzip</b>	uncompress a (gzipped) file
<b>gzip</b>	Compress a file.
<b>ps *</b>	Display running processes.
<b>tar</b>	Archive files
<b>which</b>	Find the path to a program.
<b>who</b>	See who is logged in.
<b>xargs</b>	Execute a command for each line of output.



# Summary

---

- The shell is as powerful as most programming languages
  - Actually has features that most programming languages don't
- But its limits are as important as its capabilities
  - As soon as you need functions or data structures, you should switch to **something more powerful**



# Further / Self Study

---

- Linux Shell Scripting Tutorial: A Beginner's Handbook
  - <http://www.freos.com/guides/lstt/>
- Learning the Shell
  - [http://www.linuxcommand.org/learning\\_the\\_shell.php](http://www.linuxcommand.org/learning_the_shell.php)



# Today's Agenda

---

- Further information regarding Class Project
- Finish up Advanced Shell Concepts
  - I/O Redirection & Pipes
  - Environment Variables
  - File permissions & a security issues
- Version Control Introduction
- Discuss Lab Assignment #1
- Assignment for Thursday
  - Change subversion password
  - Create and commit a text file called project.txt



# Today's Agenda

---

- Assignment for Thursday

- Change subversion password

```
[harry@nisl ~]$ htpasswd /etc/svn-auth-file harry
```

New password:

Re-type new password:

Updating password for user harry

- Check out your student repo

```
[harry@nisl ~]$ svn co http://nisl.tamu-commerce.edu/repo/csci553/harry harryrepo
```

Checked out revision 520.

- Create a text file called `project.txt`, add to repo and commit