Announcements

- Starting to wrap up the course
  - Finish any owed assignments
  - Make progress on last homework
  - Speak to me if you need help – this isn’t a paid job (might want to treat it as investment)
- Final: Monday, 12/18/2006
  - 4:10pm - 7:00pm
  - Open book, no computers
  - Will have a review before hand
  - Perl/C/C++ and anything covered in lab
Today

- Shell scripting
  - What
  - Why
  - How
  - Examples

High level review

- What is UNIX exactly?
- What are Unix flavors?
- What in the world is a command shell?
What is an Operating System

- OS?
- Where do you find them?
- What is not an OS?

Early Days - CS

- The machine
- You
- Program:
  - Bunch of wires
  - Punch cards
  - Memorized which switches to hit
- Load the “program”
- Pray like mad
  - Yay: worked
  - Hmm: froze
  - NOOOOO: crash/burn/explosion
- Debug:
  - crawl underneath and stare at switches and lights
As you have seen in the course

Not much has changed 😊

So needed a better way

OS

A program which controls hardware and software resources of a computer
- controls and allocates memory
- prioritizes system requests
- controls input and output devices
- Security level
- facilitating networking
- file management system
- Some: provide a graphical user interface for higher level functions
Brief History

- Early on, OS were specialized to hardware
  - Machine came with code libraries
  - One program at a time
  - Upgrade = new OS
    - Learn new “language”
    - Rewrite all your code
    - Painstakingly recreate all your old bugs
- Machine started to hit 100 hz!!!
- Expensive wanted to know who is using the machine to play ping pong after hours

First step

- IBM system/360
  - Family of machines, but all speak the same language
  - Backwards compatible
- CDC introduced OS for batch processing
- 1965, Bell Labs and GE
  - Multics – MIT/ARPA
    - System to support many users at the same time
    - Mainframe timesharing system
  - 1969 – Bell withdrew (too slow), but some researchers persisted on the idea of small operating system which would be portable across hardware
More history

- So first ideas coded in Assembler and B
- Rewritten in C – wanted high level code
  - First concept of software pipes
    - String together bunch of small programs to make larger
  - File system
  - Released in 1971 (time 0 in unix)
  - Released source through licensing agreements
  - Addition of TCP and specialization versions to different groups
    - Didn’t try one size fits all
  - Taught in university courses where it caught on
  - Brought to business by new graduates 😊 (early 80’s)

- So had an organized program running the show
- Need to be able to interact
- Another program!
- Command interpreter
Command shell

- Allows you to interact with the operating system
- Usually will refer to non graphical one

- Windows NT/XP:
  - Start -> run -> cmd
- Windows 98
  - Start -> run -> command
- Unix
  - Terminal / xterm
- Mac
  - terminal

Technical Details

- Shell is simply a program which takes your commands and interprets them line by line
- Usually will get to write your own in any OS course
- Many different kinds of shells
  - Mainly to confuse you 😊
- Main advantage
  - Can use build in language to write simple but powerful scripts
  - Easily edit a complicated program
- Will be “invented” in VISTA btw
Main shells (unix)

- sh – bourne shell (1978)
  - First basic one
- csh – c shell
  - C type syntax
- ksh – korn shell
  - History
- bash - Bourne again Shell
  - Superset of sh
- zsh - z shell
  - Interactive login plus shell with best of all shells
  - 1990
  - share history between all shell instances
  - Fully customizable

shell

- it can invoke Unix commands and also has its own set of commands built in
- Alogrithm:

```plaintext
while ( 1 ) {
    print prompt and wait for user to enter input;
    read input from terminal;
    parse into words;
    substitute variables;
    execute commands (execv or builtin);
}
```
- shell commands can be read:
  - from a terminal == interactive
  - from a file == shell script
    - Windows calls it batch files
      - Own syntax of course

- search path
  - the place where the shell looks for the commands it runs
  - should include standard directories:
    - /bin
    - /usr/bin
    - it should also include your current working directory (.)

- are you running the Bourne shell?
type:
echo $SHELL
- if the answer is /bin/sh, then you are
- if the answer is /bin/bash, then that’s close enough
- you can start the Bourne shell by typing sh at the UNIX prompt
- enter Ctrl-D or exit to exit the Bourne shell and go back to whatever shell you were running before...
  - Yes can recursively have a shell running a shell running a shell....you get the idea
Power of Shells

- capable of both synchronous and asynchronous execution
  - synchronous: wait for completion
  - asynchronous: in parallel with shell (runs in the background)
    - You’ve done this in the labs....

- allows control of stdin, stdout, stderr

- enables environment setting for processes (using inheritance between processes)

- sets default directory

- Problem: each command can spawn in slower thread of execution

---

Example 1

- Create a script test1.sh
  
  ```bash
  #!/bin/sh
  # This is my first script.

  echo "Hello Unix, we are in $SHELL"
  
  chmod +x test.sh
  ./test.sh
  ```
variables

- VARIABLE_NAME=value
- Command line args $1, $2 etc

comments

- single line comments only (no multi-line comments)
- line begins with # character
#!/bin/sh
# This program will read the filename
# from user input.

echo "Enter the file: "
read FILENAME
echo "Printing head of $FILENAME..."
head $FILENAME

# this prints an extra return...

echo "Printing tail of $FILENAME..."
tail $FILENAME

#!/bin/sh
# This program will print the head
# and tail of a file
# passed in on the command line.

echo "Printing head of $1..."
head $1

# this prints an extra return...

echo "Printing tail of $1..."
tail $1
Shell programming

creating your own shell scripts

- naming:
  - DON'T ever name your script (or any executable file) "test"
  - since that's a sh command

- executing
  - the notation #! inside your file tells UNIX which shell should execute the commands in your file

- example— create a file called "myscript.sh"
  
  ```sh
  #!/bin/sh
  echo hello world
  ```

- make the script executable: unix-prompt# chmod +x myscript.sh

- execute the script:
  ```sh
  ./myscript.sh
  ```

- quote (')
  'something': preserve everything literally and don't evaluate anything that is inside the quotes

- double quote ("")
  "something2": preserve most things literally, but also allow $ variable expansion (but not ‘ evaluation)

- backquote (`)'
  'something3': try to execute something as a command
Filename is t.sh
#!/bin/sh
hello=hi
echo 0=$hello
echo 1='$hello'
echo 2='$hello'
echo 3='$hello'
echo 4='$hello'
echo 5='$hello'

filename=hi
#!/bin/sh
echo "how did you get in here?"

unix$ t.sh
0=hi
1=$hello
2=hi
3=how did you get in here?
4=how did you get in here?
5=hi'

---

**Simple commands**

- sequence of words
- first word defines command
- can be combined with &&, ||, ;
  - to execute commands sequentially: cmd1; cmd2;
  - to execute a command in the background: cmd1&
  - to execute two commands asynchronously: cmd1& cmd2&
  - to execute cmd2 if cmd1 has zero exit status: cmd1 && cmd2
  - to execute cmd2 only if cmd1 has non-zero exit status: cmd1 || cmd2

- set exit status using exit command (e.g., exit 0 or exit 1)
pipes

- sequence of commands
- connected with |
- each command reads previous command’s output and takes it as input
- example:
  echo "hello world" | wc -w
  2

variables

- variables are placeholders for values
- shell does variable substitution
- $var or ${var} is the value of the variable
- assignment:
  - var=value (with no spaces before or after!)
  - let "var = value"
  - export var=value
- BUT values go away when shell is done executing
- uninitialized variables have no value
- variables are untyped, interpreted based on context
- standard shell variables:
  - ${N} = shell Nth parameter
  - $$ = process ID
  - $? = exit status
shell variables are generally not visible to programs
- environment variables are a list of name/value pairs passed to sub-processes
- all environment variables are also shell variables, but not vice versa
- show with `env` or `echo $var`
- standard environment variables include:
  - `HOME` = home directory
  - `PATH` = list of directories to search
  - `TERM` = type of terminal (vt100, ...)
  - `TZ` = timezone (e.g., US/Eastern)
Loops

- similar to C/Java constructs, but with commands
- until test-commands; do consequent-commands; done
- while test-commands; do consequent-commands; done
- for name [in words ...]; do commands; done

- also on separate lines
- break and continue control loop

- while
  i=0
  while [ $i -lt 10 ]; do
    echo "i=$i"
    ((i=$i+1)) # same as let "i=$i+1"
  done

- for
  for counter in `ls *.c`; do
    echo $counter
  done
done
if
if test-commands; then
    consequent-commands;
[elif more-test-commands; then
    more-consequents;]
[else alternate-consequents;]
fi
- colon (:) is a null command
- example
  #!/bin/sh
  if expr $TERM = "xterm"; then
    echo "hello xterm";
  else
    echo "something else";
  fi

comparisons
- eq  equal to
- ne  not equal to
- lt  less than
- le  less than or equal to
- gt  greater than
- ge  greater than or equal to
Files

- s  file exists and is not empty
- f  file exists and is not a directory
- d  directory exists
- x  file is executable
- w  file is writable
- r  file is readable

#!/bin/sh

# Prompt for a user name...
# Prompt for a user name...
echo "Please enter your age:"
read AGE

if [ "$AGE" -lt 20 ] || [ "$AGE" -ge 50 ]; then
  echo "Sorry, you are out of the age range."
elif [ "$AGE" -ge 20 ] && [ "$AGE" -lt 30 ]; then
  echo "You are in your 20s"
elif [ "$AGE" -ge 30 ] && [ "$AGE" -lt 40 ]; then
  echo "You are in your 30s"
elif [ "$AGE" -ge 40 ] && [ "$AGE" -lt 50 ]; then
  echo "You are in your 40s"
fi
case test-var in
  value1) consequent-commands;;
  value2) consequent-commands;;
  *) default-commands;
esac

- pattern matching:
  - ?) matches a string with exactly one character
  - ?*) matches a string with one or more characters
  - [yY][yY][eE][sS]) matches y, Y, yes, YES, yES...
  - /*/*[0-9]) matches filename with wildcards like /xxx/yyyy/zzz3
  - notice two semi-colons at the end of each clause
  - stops after first match with a value
  - you don’t need double quotes to match string values!

example

#!/bin/sh

case "\$TERM" in
  xterm) echo "hello xterm";;
  vt100) echo "hello vt100";;
  *) echo "something else";;
esac
biggest difference from traditional programming languages

shell substitutes and executes

order:
  - brace expansion
  - tilde expansion
  - parameter and variable expansion
  - command substitution
  - arithmetic expansion
  - word splitting
  - filename expansion

Command subing
  - replace $(command) or 'command' by stdout of executing command
  - can be used to execute content of variables:

```bash
unix$ x=ls
unix$ ls
myfile.c
a.out
unix$ echo $x
ls
unix$ echo 'ls'
myfile.c
a.out
unix$ echo 'ls'
ls
unix$ echo '$x'
ls
unix$ echo $(ls)
myfile.c
a.out
unix$ echo $(x)
sh: x: command not found
unix$ echo $(ls)
myfile.c
a.out
unix$ echo $x
sh: x: command not found
unix$ echo $x
myfile.c
a.out
```
Filename expansion

- any word containing *?[{} is considered a pattern
- * matches any string
- ? matches any single character
- [...] matches any of the enclosed characters

```
unix$ ls
myfile.c
a.out
a.b
unix$ ls a*
a.out
a.b
unix$ ls a?
a: No match.
unix$ ls a.*
a.out
a.b
unix$ ls a.?
a.b
unix$ ls a.???
a.out
unix$ ls [am].b
a.b
```

redirection

- stdin, stdout and stderr may be redirected
- < redirects stdin (0) to come from a file
- > redirects stdout (1) to go to file
- >> appends stdout to the end of a file
- &> redirects stderr (2)
- >& redirects stdout and stderr, e.g.: 2>&1 sends stderr to the same place that stdout is going
- << gets input from a here document, i.e., the input is what you type, rather than reading from a file

```
```
Built in commands

- alias, unalias — create or remove a pseudonym or shorthand for a command or series of commands
- jobs, fg, bg, stop, notify — control process execution
- command — execute a simple command
- cd, chdir, pushd, popd, dirs — change working directory
- echo — display a line of text
- history, fc — process command history list
- set, unset, setenv, unsetenv, export — shell built-in functions to determine the characteristics for environmental variables of the current shell and its descendents
- getopts — parse utility options
- hash, rehash, unhash, hashstat — evaluate the internal hash table of the contents of directories
- kill — send a signal to a process

- pwd — print name of current/working directory
- shift — shell built-in function to traverse either a shell’s argument list or a list of field-separated words
- readonly — shell built-in function to protect the value of the given variable from reassignment
- source — execute a file as a shell script
- suspend — shell built-in function to halt the current shell
- test — check file types and compare values
- times — shell built-in function to report time usages of the current shell
- trap, onintr — shell built-in functions to respond to (hardware) signals
- type — write a description of command type
- typeset, whence — shell built-in functions to set/get attributes and values for shell variables and functions
- `limit, ulimit, unlimit` — set or get limitations on the system resources available to the current shell and its descendents
- `umask` — get or set the file mode creation mask

---

**More programs you might like**

- `cal`  
  - Prints a calendar

```
bash-2.05$ cal 2 2004
February 2004
Su Mo Tu We Th Fr Sa
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29
```
Usage stuff

- **df**
  ```bash
  bash-2.05$ df -h
  Filesystem Size  Used Avail Use% Mounted on
  /dev/hda3 197M  157M   31M  84% /
  /dev/hda7  296M   65k  280M   1% /tmp
  /dev/hda5  2.4G  2.0G  385M  84% /usr
  ```

- **du**
  ```bash
  bash-2.05$ du -ch code2
  48k   code2/ai1
  56k   code2
  56k   total
  ```

- **quota**

Useful programs

- Here is a list of useful utilities standard in many unix like shells
Useful tools & commands

- wc
- grep
- cut
- cat
- sort
- sed
- ps
- who
- finger
- kill
- touch
- Test
- diff

**WC**

- counts the number of characters/words/lines in its input
  - input can be a file or a piped command (see below)

  **filename = "hello.dat"**
  hello
  world
  **usage:**
  unix-prompt$ wc hello.dat
  2 2 12 hello.dat
  unix-prompt$ wc -l hello.dat
  2 hello.dat
  unix-prompt$ wc -c hello.dat
  12 hello.dat
  unix-prompt$ wc -w hello.dat
  2 hello.dat
Global Regular Expression Parser
GREP

- one of the most useful tools in unix – make sure it's your friend
- three standard versions:
  - plain old grep
  - extended grep: egrep
  - fast grep: fgrep
- used to search through files for ... regular expressions!
- prints only lines that match the given pattern
- a kind of filter
- BUT it's line oriented

- input can be one or more files or can be piped into grep
- examples:
grep "^[aeiou]" myfile
ls -l | grep t
- useful options:
  -i ignore case
  -w match pattern as a word
  -l return only the filename if there's a match
  -v reverse the normal action (i.e., return what doesn't match)
**examples:**
grep -i "^[aeiou]$" myfile
grep -v "^[aeiou]$" myfile
grep -iv "^[aeiou]$" myfile

- how do you list all lines containing a digit?
- how do you list all lines containing a 5?
- how do you list all lines containing a 0?
- how do you list all lines containing 50?
- how do you list all lines containing a 5 and an 0?

---

**cut**

- unix command: extracts portions of each line from input/file to stdout

- Can cut file according to delimiters (fields) and characters

- syntax: cut <-c|f> <-d>

- note that c and f start with 1; default delimiter is TAB

- Example
  - ls -la | cut -c1-12
  - cut -c12-24,34-39
cat

- Concatenate files and print to standard out

- Easy way to pipe the contents of a file or files to another command

sort

- sorts lines of input

- input can be a file or a piped command (see below)

- three modes: sort, check (sort -c), merge (sort -m)

- syntax: sort <-t> <-n> <-r> <-o> POS1 -POS2+
  -n Numeric
  -M sort months Jan < Dec
  -f ignore case
  -r reverse the sort

- note that POS starts with 0; default delimiter is whitespace
Question?

So what does:

```
ls -la | cut -c35-42 | sort -n
```

What is this?

```bash
#!/bin/sh
foo() {
  if [ "$1" -gt "1" ]; then
    i='expr $1 - 1'
    j='foo $i'
    k='expr $1 \* $j'
    echo $k
  else
    echo 1
  fi
}

while :
do
  echo "Enter a number:"
  read x
  foo $x
done
```
sed – stream editor

- stream editor - does not actually change the file it "edits"
- commands are implicitly global
- example: substitute all A for B:
  - `sed 's/A/B/' myfile`
  - `cat myfile | sed 's/A/B/'`
- use the -e option to specify more than one command at a time:
  - `sed -e 's/A/B/' -e 's/C/D/' myfile`
- pipe output to a file in order to save it:
  - `sed -e 's/A/B/' -e 's/C/D/' myfile >mynewfile`

sed

- sed can specify an address of the line(s) to affect
- if no address is specified, then all lines are affected
- if there is one address, then any line matching the address is affected
- if there are two (comma separated) addresses, then all lines between the two addresses are affected
- if an exclamation mark (!) follows the address, then all lines that DON'T match the address are affected
- addresses are used in conjunction with commands
- examples (using the delete (d) command):
  - `sed '$d' myfile`
  - `sed '/^$/d' myfile`
  - `sed '1,/under/d' myfile`
  - `sed '/over/,/under/d' myfile`
- order of commands is important
- input is line oriented
- all editing commands are applied to each line, one at a time
- then next line is read and editing commands are applied to that line
- etc

For example:
```
$ sed -e 's/pig/cow/' -e 's/cow/horse' myfile
```
- What does this do?

- Regular expression like grep
- Except forward slash
- delimiter is slash (/)
- backslash (escape) it if it appears in the command, e.g.:
```
$ sed 's/\usr/bin/\usr/etc/\'
```
myfile
- meta-character ampersand (&) represents the extent of the pattern matched
- example:
  sed 's/[0-9]/#&/' myfile
- what does this do?

- you can also save portions of the matched pattern:
  sed 's/\([0-9]\)/#\1/' myfile
  sed 's/\([0-9]\)\([0-9]\)/#\1-\2/' myfile

- transformation command: y
- example:
  sed 'y/ABC/abc' myfile
- print command: `p`

- example:
  ```
  sed '/begin/./end/p' myfile
  sed -n '/begin/./end/p' myfile
  ```

- what do the following sed commands do?
  ```
  sed 's/xx/yy' myfile
  sed '/BSD/d' myfile
  sed '/^BEGIN/./^END/p' myfile
  ```

- how do you change the content of all your html files to lowercase?

- how do you change all the html commands to lowercase?