Objectives

- Bash scripting

Review

- What are special characters for regular expressions and what do they mean?
- Which command should you use for fast, enhanced searching with regular expressions?
- What is backreferencing?
- Which command should you use if you want backreferencing?
- What is the syntax for those commands?

Shell Scripts

- ***Script***: a shell program
- Tool for building applications by "gluing together" system calls, tools, utilities, and compiled binaries
- Just about everything we've done so far is available for use in a script
  - Adds even more
- Good for repetitive tasks that don't require a more structured programming language

Shell Scripting vs. [C/Python/Java] Programming

Advantages
Easy to work with/use other programs
Easy to work with directories, files
Easy to work with strings (easier than C, at least)
Good for prototyping

Shell Scripting vs. [C/Python/Java] Programming

Advantages	Disadvantages
Easy to work with/use other programs	Slower
Easy to work with directories, files	Not well-suited for algorithms and data structures
Easy to work with strings (easier than C, at least)	
Good for prototyping	

Scripts won't be long

In some ways, we'll love it; in some ways, we'll hate it.

The C Shell

- C-like syntax (uses **{ }**'s)
- **Inadequate for scripting**
  - Poor control over file descriptors
  - Difficult quoting `"I say \"hello\""` doesn't work
  - Can only trap SIGINT
  - Can't mix flow control and commands
- But has some nice interactive features
  - Job control
  - Command history
  - Command line editing, with arrow keys (**tcsh**)

http://www.faqs.org/faqs/unix-faq/shell/csh-whynot

The Bourne Shell (sh)

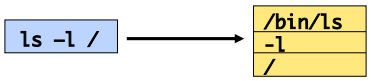
- Slight differences on various systems
- Evolved into standardized POSIX shell
- Scripts will also run with **ksh**, **bash** (Bourne-again shell)
- Influenced by ALGOL

Just about everything we've learned so far can be used in scripts…

REVIEW

Review: Simple Commands

- ***simple command***: sequence of non blanks arguments separated by blanks or tabs.
- 1st argument (numbered zero) usually specifies the name of the command to be executed.
- Any remaining arguments:
  - Are passed as arguments to that command.
  - Arguments may be filenames, pathnames, directories or special options



Review: Background Commands

- Any command ending with "&" is run in the background

```
firefox &
```

- `wait` will block until the command finishes
  - If give a parameter *n*, *n* may be a process ID or a job specification
    - if a job spec, all processes in that job's pipeline are waited for
  - If n is not given, all currently active child processes are waited for and the return status is zero.

Review: Complex Commands

- The shell's power is in its ability to hook commands together
- We've seen one example of this so far with pipelines:

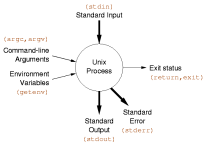
```
cut -d: -f2 /etc/passwd | sort | uniq
```

What does this do?

- We will see others

Review: UNIX Programs

- **Means of input**:
  - Program arguments [control information]
  - Environment variables [state information]
  - Standard input [data]
- **Means of output**:
  - Return status code [control information]
  - Standard out [data]
  - Standard error [error messages]



Review: Redirection of input/ouput

- Redirection of output: >
  - `example:$ ls -l > my_files`
- Redirection of input: <
  - `example: $ cat <input.data`
- Append output: >>
  - `example: $ date >> logfile`
- Arbitrary file descriptor redirection: fd>
  - `example: $ ls –l 2> error_log`

Multiple Redirection

- `cmd 2>file`
  - send standard error to file
  - standard output remains the same
- `cmd > file 2>&1`
  - send both standard error and standard output to file
- `cmd > file1 2>error_log`
  - send standard output to file1
  - send standard error to error_log

SHELL SCRIPTING

Types of Commands

All behave the same way

- Programs
  - Most that are part of the OS in **`/bin`**
- Built-in commands
- Functions
- Aliases

Shell Scripts

- A shell script is a regular **text** file that contains shell or UNIX commands
- Kernel uses the *first line* of script to determine which shell script to use
  - `#!pathname-of-shell`
    - Kernel invokes pathname and sends the script as an argument to be interpreted
  - If `#!` is not specified, the current shell assumes it is a script in its own language
    - Can lead to problems

Simple Example

```
#!/bin/sh
echo Hello World
```

Which shell to use

Command to execute

`echo` – like a print statement

Note: Look at the available shells by executing

```
ls -l /bin/*sh
```

What do you notice about the shells?

Invoking a Script

- A script can be invoked as:
  - `sh scr_name [ arg … ]`
  - `sh < scr_name [ args … ]`
  - `path/scr_name [ arg …]`
    - Before running it, it must have execute permission:
      - `chmod +x scr_name`

Where sh is whatever shell you want

We'll typically use either the 1st or 3rd execution option and we'll use the bash shell

Your First Script

- Write a script that
  - Shows the time and date
  - Lists all logged-in users
  - Saves the output into a logfile
- Build in pieces
- Execute and test your script
  - Verify the output in the logfile

Built-in Commands

- Built-in commands are internal to the shell and do not create a separate process
- Commands are built-in because:
  - They are intrinsic to the language (`exit`)
  - They produce side effects on the current process (`cd`)
  - They perform faster
    - No fork/exec
- Special built-ins
  - `: . break continue eval exec export exit readonly return set shift trap unset`

Important Built-in Commands

exec	Replaces shell with program
cd	Change working directory
shift	Rearrange positional parameters
set	Set positional parameters
wait	Wait for background process to exit
umask	Change default file permissions
exit	Quit the shell
eval	Parse and execute string

Important Built-in Commands

time	Run command and print times
export	Put variable into environment
trap	Set signal handlers
continue	Continue in loop
break	Break in loop
return	Return from function
:	True
.	Read file of commands into current shell

Comments

- Comments begin with an **#**
- Comments end at the end of the line
- Comments can begin whenever a token begins
- Our text editors should help you with syntax highlighting
- Examples:

```
# This is a comment
# and so is this
grep foo bar # this is a comment
grep foo bar# this is not a comment
```

Add a comment at 2nd line in your script that lists you as author

Variables

- To set:
  `name=value` ← Notice no spaces around =
  - Variables are *untyped*
- Read: `$var`
- Variables can be local or environment
  - Environment variables are part of UNIX and can be accessed by child processes
- Turn local variable into environment var:
  `export variable`

Variable Example

```
#!/bin/sh

MESSAGE="Hello World"
echo $MESSAGE
echo '$MESSAGE'
echo "$MESSAGE"
```

Prints variable
Prints literally
Prints variable

variable.sh

Environmental Variables

Name	Meaning
$HOME	Absolute pathname of your home directory
$PATH	A list of directories to search for
$MAIL	Absolute pathname to mailbox
$USER	Your user name
$SHELL	Absolute pathname of login shell
$TERM	Type of terminal
$PS1	Prompt

Using Environment Variables

```
#!/bin/bash

echo I am $USER
echo "I live at $HOME"
```

Both statements would work either with or without quotes

env_var.sh

Parameters

- A parameter is one of the following:
  - A *positional* parameter, starting from 0
  - A *special* parameter
- To get the value of a parameter: **`${param}`**
  - Can be part of a word (**`abc${foo}def`**)
  - Works within double quotes
- The **`{}`** can be omitted for simple variables, special parameters, and single digit positional parameters

Positional Parameters

- The arguments to a shell script
  - `$0, $1, $2, $3 …`
  - Parameter 0 is the name of the shell or the shell script
- The arguments to a shell *function*
- Arguments to the `set` built-in command
  - `set this is a test`
    - `$1=this, $2=is, $3=a, $4=test`
- Manipulated with `shift`
  - `shift 2`
    - `$1=a, $2=test`

Example with Parameters

Script

```
#!/bin/sh

# Parameter 1: string
# Parameter 2: file
grep $1 $2 | wc -l
```

Invocation:

```
$ countlines ing /usr/share/dict/words
30415
```

 countlines

Special Parameters

Parameter	Meaning
$#	Number of positional parameters
$-	Options currently in effect
$?	Exit value of last executed command
$$	Process number of current process
$!	Process number of background process
$*	All arguments on command line from 1 on
"$@"	All arguments on command line Individually quoted "$1" "$2" …; good if parameters contain spaces

 countlines_print

Special Characters

- The shell processes the following characters specially unless quoted:
  - `| & ( ) < > ; " ' $ ` space tab newline`
- The following are special whenever patterns are processed:
  - `* ? [ ]`
- The following are special at the beginning of a word:
  - `# ~`
- The following is special when processing assignments:
  - `=`

Here Documents

- Shell provides alternative ways of supplying standard input to commands (an anonymous file)
- Shell allows in-line input redirection using `<<` called ***here documents***
- Syntax:

```
command [arg(s)] << arbitrary-delimiter
      command input
       :
       :
      arbitrary-delimiter
```

- `arbitrary-delimiter` should be a string that does not appear in text

Here Document Example

```
#!/bin/sh

mail –s "Groceries" sprenkles@wlu.edu << END
Don't forget your grocery list
Eggs
Milk
Bread
END
```

 groceries.sh

Command Substitution: ``

- Used to turn the output of a command into a string
- *Used to create arguments or variables*

```
$ date
Wed Apr 29 10:55:51 EDT 2009
$ NOW=`date`
$ echo $NOW
Wed Apr 29 10:55:59 EDT 2009
$ PATH=`myscript`:$PATH
```

Compound Commands

- Multiple commands
  - Separated by semicolon or newline
- Command groupings
  - pipelines
- Subshell
  `( command1; command2 ) > file`
- Boolean operators
- Control structures

Boolean Operators

- Exit value of a program is a number
  - 0 means success
  - anything else is a failure code
- `cmd1 && cmd2`
  - executes cmd2 if cmd1 is successful
- `cmd1 || cmd2`
  - executes cmd2 if cmd1 is not successful

Send output to black hole (Can't be read)

```
$ ls bad_file > /dev/null && date
$ ls bad_file > /dev/null || date
Wed Apr 22 07:43:23 2009
```


Control Structures

```
if expression
then
        command1
else
        command2
fi
```


What is an expression?

- Any UNIX command. Evaluates to true if the exit code is 0, false if the exit code > 0
- Special command **`/bin/test`** does most common expressions:
  - String compare
  - Numeric comparison
  - Check file properties
- **`[ ]`** often a built-in version of **`/bin/test`** for syntactic sugar

Examples

```
if test "$USER" = "sprenkle"
then
        echo "I know you"
else
        echo "I don't know you"
fi
```

know.sh

```
if [ -f /tmp/stuff ] && [ `wc -l /tmp/stuff | cut -f1 -d " "` -
gt 10 ]
then
        echo "The file has more than 10 lines in it"
else
        echo "The file is nonexistent or small"
fi
```

filesize.sh

`test` Summary

- String based tests

`-z string`	Length of string is 0
`-n string`	Length of string is not 0
`string1 = string2`	Strings are identical
`string1 != string2`	Strings differ
`string`	string is not NULL

- Numeric tests

`int1 -eq int2`	First int equal to second
`int1 -ne int2`	First int not equal to second
`-gt, -ge, -lt, -le`	greater, greater/equal, less, less/equal

test Summary

- File tests

-r file	File exists and is readable
-w file	File exists and is writable
-f file	File is regular file (exists)
-d file	File is directory
-s file	File exists and is not empty

- Logic

!	Negate result of expression
-a, -o	And operator, or operator
(expr)	Groups an expression

What does this code do?

```
ARGS=1      # Number of arguments expected
# Exit value if incorrect number of args passed.
E_BADARGS=65

test $# -lt $ARGS && echo "Usage: `basename $0` <arg1>" && \
exit $E_BADARGS
```

- Add appropriate code to countlines

Arithmetic

- Use external command /bin/expr
- expr expression
  - Evaluates expression and sends the result to standard output
  - Yields a numeric or string result

```
expr 4 "*" 12
expr "(" 4 + 3 ")" "*" 2
```

Need to quote the * b/c shell interprets it

  - Particularly useful with command substitution

```
X=`expr $X + 2`
```

arith_operators.sh

Double parentheses and the let statement

- Double parentheses

```
z=$(($z+3))
z=$((z+3))
```

- Let statement

```
let z=z+3
let "z += 3"
```

Quotes permit the use of spaces in variable assignment

let.sh

Control Structures Summary

- if … then … fi
- while … done
- until … do … done
- for … do … done
- case … in … esac

for loops

```
for var in list
do
      command
done
```

- Examples:

```
sum=0
for var in "$@"
do
       sum=`expr $sum + $var`
done
echo The sum is $sum
```

sum_params.sh

```
for file in *.c
do
      echo "We have $file"
done
```

for_file.sh
for_params.sh

Practice

- Write a script that copies all files in a directory into a backup subdirectory
  - Takes as a parameter the directory

Assignment 5: Bash Scripting

- Due Friday
- Write in small parts and test
  - Remember you're learning a new language!!
- Comment each script
  - Every script should contain your name and a high-level description
    - Helpful to refer to later
- Friday: advanced bash scripting