Shell Scripting
and
Regular Expressions

Week 2

The Shell and OS

- The shell is the user's interface to the OS
- From it you run programs.
- Common shells
  - Bash, csh, sh, tcsh
- Allow more complex functionality then just interacting with OS directly
  - Tab complete, easy redirection

Scripting Vs Compiled Languages

- Compiled Languages
  - Ex: C/C++
  - Programs are translated from their original source code into object code that is executed by hardware
  - Efficient
  - Work at low level, dealing with bytes, integers, floating points, etc
- Scripting languages
  - Interpreted
  - Interpreter reads program, translates it into internal form, and execute programs

Example

Example:

```
$ who | grep betsy  Where is betsy?
betsy pts/3 Dec 27 11:07 (flags-r-us.example.com)
```

Script:

```
#! /bin/sh
# finduser --- see if user named by first argument is logged in
who | grep $1
```

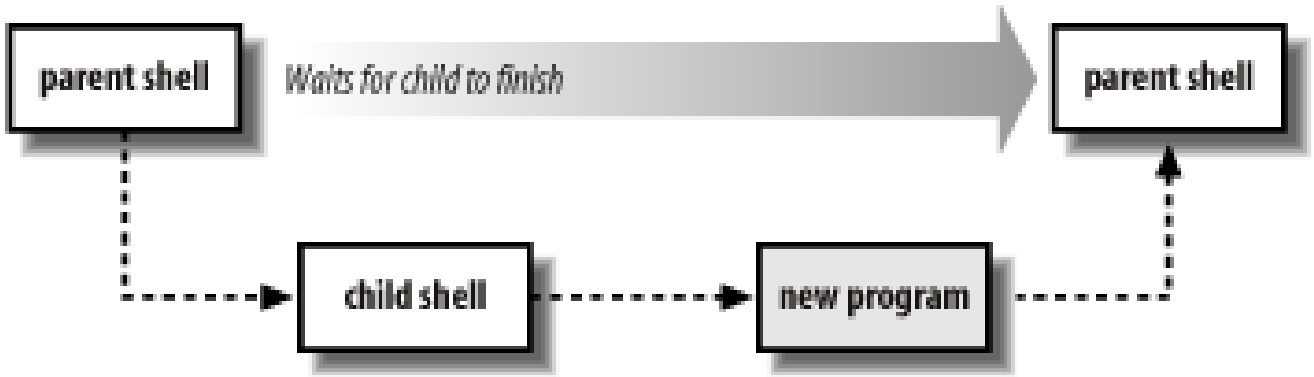
Run it:

```
$ chmod +x finduser     Make it executable
$ ./finduser betsy  Test it: find betsy
betsy pts/3 Dec 27 11:07 (flags-r-us.example.com)
$ ./finduser benjamin  Now look for Ben
 benjamin dtlocal Dec 27 17:55 (kites.example.com)
```

Self-Contained Scripts: The #! First Line

- A shell script file is just a file with shell commands
- When shell script is executed a new child “shell” process is spawned to run it
- The first line is used to state which child “shell” to use

```
#! /bin/csh -f
#! /bin/awk -f
#! /bin/sh
#! /bin/bash
```



Ubuntu Shell Scripting

- Ubuntu 6.01+ uses by default “dash” shell which is POSIX compliant
- /bin/sh is a link to /bin/dash
- “dash” and “bash” should not have any differences for our use
- Bash tutorial http://linuxconfig.org/Bash_scripting_Tutorial

Variables

- Declared using =
  - Var = “hello”
- Referenced using $
  - echo $Var

```
#!/bin/sh
STRING="HELLO WORLD!!!"
echo $STRING
```

POSIX Built-in Shell Variables

Variable	Meaning
#	Number of arguments given to current process.
@	Command-line arguments to current process. Inside double quotes, expands to individual arguments.
*	Command-line arguments to current process. Inside double quotes, expands to a single argument.
- (hyphen)	Options given to shell on invocation.
?	Exit status of previous command.
$	Process ID of shell process.
0 (zero)	The name of the shell program.
!	Process ID of last background command. Use this to save process ID numbers for later use with the *wait* command.
ENV	Used only by interactive shells upon invocation; the value of $ENV is parameter-expanded. The result should be a full pathname for a file to be read and executed at startup. This is an XSI requirement.
HOME	Home (login) directory.
IFS	Internal field separator; i.e., the list of characters that act as word separators. Normally set to space, tab, and newline.

POSIX Built-in Shell Variables

Variable	Meaning
LC_ALL	Name of current locale; overrides LANG and the other LC_* variables.
LC_COLLATE	Name of current locale for character collation (sorting) purposes.
LC_CTYPE	Name of current locale for character class determination during pattern matching.
LC_MESSAGES	Name of current language for output messages.
LINENO	Line number in script or function of the line that just ran.
NLSPATH	The location of message catalogs for messages in the language given by $LC_MESSAGES (XSI).
PATH	Search path for commands.
PPID	Process ID of parent process.
PS1	Primary command prompt string. Default is "$ ".
PS2	Prompt string for line continuations. Default is "> ".
PS4	Prompt string for execution tracing with set -x. Default is "+ ".
PWD	Current working directory.

Accessing Arguments

- Positional parameters represent a shell script's command-line arguments
- For historical reasons, enclose the number in braces if it's greater than 9

```
#!/bin/sh
#test script
echo first arg is $1
echo tenth arg is ${10}

./test hello
first arg is hello
```

If statements

- If statements use the test command or [ ]
- “man test” to see the expressions that can be created

```
#!/bin/bash
if [ 5 –gt 1 ];
then
      echo “5 greater than 1"
else
      echo “not possible"
fi
```

Exit: Return value

Check exit status of last command that ran with $?

Value	Meaning
0	Command exited successfully.
> 0	Failure during redirection or word expansion (tilde, variable, command, and arithmetic expansions, as well as word splitting).
1-125	Command exited unsuccessfully. The meanings of particular exit values are defined by each individual command.
126	Command found, but file was not executable.
127	Command not found.
> 128	Command died due to receiving a signal.

Quotes

- Three kinds of quotes
  - Backticks ``
    - Expand as shell commands
      - `temp=`ls` ; echo $temp`
    - Same as
      - `temp=$(ls); echo $temp`
  - Single quotes ‘ ‘
    - Do not expand at all, literal meaning
      - `temp=‘$hello$hello’ ; echo $temp`
  - Double quotes “ “
    - Almost like single quotes but expand backticks, $, \ characters

Loops

- While loop

```
#!/bin/sh
COUNT=6
while [ $COUNT -gt 0 ];
do
        echo Value of count is: $COUNT
        let COUNT=COUNT-1
done
```

  - The “let” command is used to do arithmetic
- For loop

```
#!/bin/sh
temp=`ls`
for f in $temp;
do
        echo $f
done
```

  - f will refer to each word in $temp escaped by whitespace

Output Using Echo or Printf

Using **echo** for printing

```
$ echo "Hello \n world"
Hello \n world

$ echo -e "Hello\nworld"
Hello
world
```

Using **printf**

```
$ printf "Hello \n world"
Hello
world
```

Stdout, Stdin, Stderr

- Most programs by default
  - Output to stdout, which is terminal
  - Takes input from stdin, which is terminal
  - Output errors to stderr, which is terminal

Redirection and Pipelines

- Use *program < file* to make *program*'s stdin to be *file*:

```
cat < file
```

- Use *program > file* to make *program*'s stdout be *file*:

```
cat < file > file2
```

- Use *program 2> file* to make *program*'s stderr be *file*:

```
cat < file 2> file2
```

- Use *program >> file* to append stdout to file
- Use *program1 | program2* to make the stdout of *program1* become the stdin of *program2*.

```
cat < file | sort > file2
```

Simple Execution Tracing

- To get shell to print out each command as it's execute, precede it with "+"
- You can turn execution tracing within a script by using:

  `set -x`: to turn it on

  `set +x`: to turn it off

Searching for Text

- grep: Uses basic regular expressions (BRE)
- egrep: Extended grep that uses extended regular expressions (ERE)
- Fgrep: Fast grep that matches fixed strings instead of regular expressions.

Simple grep

```
$ who                                                    Who is logged on
tolstoy tty1 Feb 26 10:53
tolstoy pts/0 Feb 29 10:59
tolstoy pts/1 Feb 29 10:59
tolstoy pts/2 Feb 29 11:00
tolstoy pts/3 Feb 29 11:00
tolstoy pts/4 Feb 29 11:00
austen pts/5 Feb 29 15:39 (mansfield-park.example.com)
austen pts/6 Feb 29 15:39 (mansfield-park.example.com)

$ who | grep -F austen                        Where is austen logged on?
austen pts/5 Feb 29 15:39 (mansfield-park.example.com)
austen pts/6 Feb 29 15:39 (mansfield-park.example.com)
```

Regular Expressions

- Notation that lets you search for text that fits a particular criterion, such as “starts with the letter a”
- http://regexpal.com/ to test your regex expressions
- Simple regex tutorial http://www.icewarp.com/support/online_help/203030104.htm

Regular expressions

Character	BRE / ERE	Meaning in a pattern
\	Both	Usually, turn off the special meaning of the following character. Occasionally, enable a special meaning for the following character, such as for \(...\) and \{...\}.
.	Both	Match any single character except NUL. Individual programs may also disallow matching newline.
*	Both	Match any number (or none) of the single character that immediately precedes it. For EREs, the preceding character can instead be a regular expression. For example, since . (dot) means any character, .* means "match any number of any character." For BREs, * is not special if it's the first character of a regular expression.
^	Both	Match the following regular expression at the beginning of the line or string. BRE: special only at the beginning of a regular expression. ERE: special everywhere.

Regular Expressions (cont'd)

$	Both	Match the preceding regular expression at the end of the line or string. BRE: special only at the end of a regular expression. ERE: special everywhere.
[...]	Both	Termed a bracket expression, this matches any one of the enclosed characters. A hyphen (-) indicates a range of consecutive characters. (Caution: ranges are locale-sensitive, and thus not portable.) A circumflex (^) as the first character in the brackets reverses the sense: it matches any one character not in the list. A hyphen or close bracket (]) as the first character is treated as a member of the list. All other metacharacters are treated as members of the list (i.e., literally). Bracket expressions may contain collating symbols, equivalence classes, and character classes (described shortly).
\{*n*,*m*\}	BRE	Termed an *interval expression*, this matches a range of occurrences of the single character that immediately precedes it. \{*n*\} matches exactly n occurrences, \{*n*,\} matches at least n occurrences, and \{*n*,*m*\} matches any number of occurrences between n and m. n and m must be between 0 and RE_DUP_MAX (minimum value: 255), inclusive.
\(\)	BRE	Save the pattern enclosed between \(and \) in a special *holding space*. Up to nine subpatterns can be saved on a single pattern. The text matched by the subpatterns can be reused later in the same pattern, by the escape sequences \1 to \9. For example, **\(ab\).*\1** matches two occurrences of ab, with any number of characters in between.

Regular Expressions (cont'd)

\n	BRE	Replay the nth subpattern enclosed in \(and \) into the pattern at this point. n is a number from 1 to 9, with 1 starting on the left.
{n,m}	ERE	Just like the BRE \{*n*,*m*\} earlier, but without the backslashes in front of the braces.
+	ERE	Match one or more instances of the preceding regular expression.
?	ERE	Match zero or one instances of the preceding regular expression.
|	ERE	Match the regular expression specified before or after.
( )	ERE	Apply a match to the enclosed group of regular expressions.

Examples

Expression	Matches
tolstoy	The seven letters tolstoy, anywhere on a line
^tolstoy	The seven letters tolstoy, at the beginning of a line
tolstoy$	The seven letters tolstoy, at the end of a line
^tolstoy$	A line containing exactly the seven letters tolstoy, and nothing else
[Tt]olstoy	Either the seven letters Tolstoy, or the seven letters tolstoy, anywhere on a line
tol.toy	The three letters tol, any character, and the three letters toy, anywhere on a line
tol.*toy	The three letters tol, any sequence of zero or more characters, and the three letters toy, anywhere on a line (e.g., toltoy, tolstoy, tolWHOtoy, and so on)

POSIX Bracket Expressions

Class	Matching characters	Class	Matching characters
[:alnum:]	Alphanumeric characters	[:lower:]	Lowercase characters
[:alpha:]	Alphabetic characters	[:print:]	Printable characters
[:blank:]	Space and tab characters	[:punct:]	Punctuation characters
[:cntrl:]	Control characters	[:space:]	Whitespace characters
[:digit:]	Numeric characters	[:upper:]	Uppercase characters
[:graph:]	Nonspace characters	[:xdigit:]	Hexadecimal digits

Matching Multiple Characters with One Expression

*	Match zero or more of the preceding character
\{*n*\}	Exactly n occurrences of the preceding regular expression
\{*n*,\}	At least n occurrences of the preceding regular expression
\{*n*,*m*\}	Between n and m occurrences of the preceding regular expression

Anchoring text matches

Pattern	Text matched (in bold) / Reason match fails
ABC	Characters 4, 5, and 6, in the middle: abc**ABC**defDEF
^ABC	Match is restricted to beginning of string
def	Characters 7, 8, and 9, in the middle: abcABC**def**DEF
def$	Match is restricted to end of string
[[:upper:]]\{3\}	Characters 4, 5, and 6, in the middle: abc**ABC**defDEF
[[:upper:]]\{3\}$	Characters 10, 11, and 12, at the end: abcDEFdef**DEF**
^[[:alpha:]]\{3\}	Characters 1, 2, and 3, at the beginning: **abc**ABCdefDEF

sed

- Now you can extract, but what if you want to replace parts of text?
- Use sed!

```
sed 's/regExpr/replText/'
```

- Example

```
sed 's/:.*//' /etc/passwd
```

Remove everything after the first colon

Text Processing Tools

- sort: sorts text
- wc: outputs a one-line report of lines, words, and bytes
- head: extract top of files
- tail: extracts bottom of files

Lab Hints

- `sed '/patternstart/,/patternstop/d'`
  - To delete all lines from patternstart to patternstop. Works across multiple lines.
- `od -c webpage.htm`
  - to see the ascii characters
  - the hawaiian words html page uses \r and \n to do new lines
- `tr -d '[:blank:]'`
  - you can delete blank white spaces such as tab or space
- `tr -s`
  - to squeeze multiple new lines into one
- `IFS=$'\n'`
  - to split at newlines

HW SampleCode

```
#!/bin/sh

dir=$1
RESULT=`ls -a $dir`
declare -a ARRAY
let count=0;
for FILE in $RESULT
do
    if [ -f "$dir/$FILE" ];
    then
        echo "$dir/$FILE is regular file"
        ARRAY[$count]="$dir/$FILE";
        let count=count+1;
    else
        echo "$dir/$FILE is not regular file"
    fi
done
echo $count
```