Lecture 5

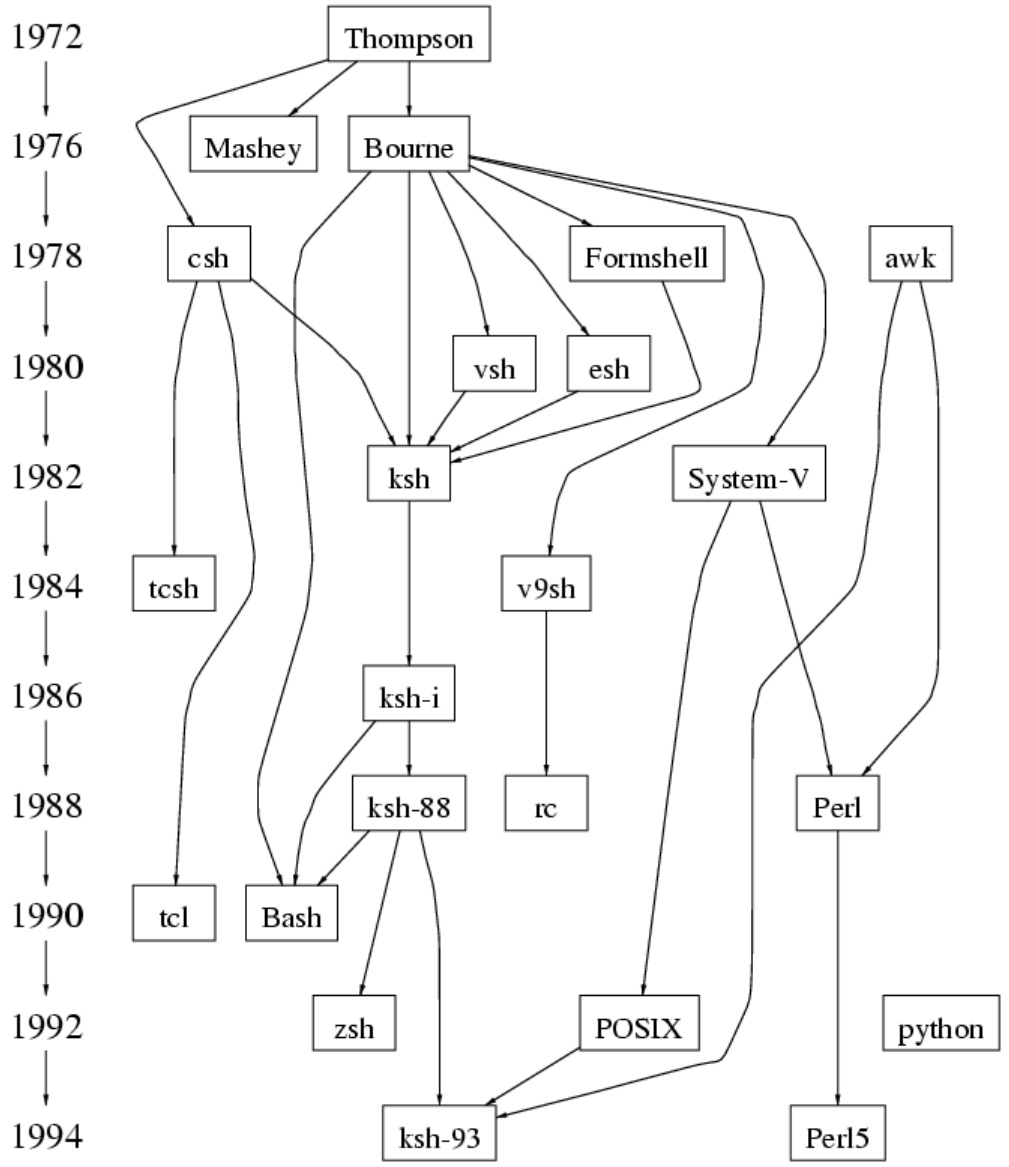
Shell Scripting

What is a shell?

- The user interface to the operating system
- Functionality:
  - Execute other programs
  - Manage files
  - Manage processes
- Full programming language
- A program like any other
  - This is why there are so many shells

Shell History

- There are many choices for shells
- Shell features evolved as UNIX grew



Shell Scripts

- A shell script is a regular text file that contains shell or UNIX commands
  - Before running it, it must have execute permission:
    - **`chmod +x filename`**
- A script can be invoked as:
  - **`ksh name [ arg … ]`**
  - **`ksh < name [ args … ]`**
  - **`name [ arg …]`**

Shell Scripts

- When a script is run, the **kernel** determines which shell it is written for by examining the first line of the script
  - If 1st line starts with **`#!`*`pathname-of-shell`***, then it 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
    - leads to problems

Simple Example

```
#!/bin/sh

echo Hello World
```

Scripting vs. C Programming

- Advantages of shell scripts
  - Easy to work with other programs
  - Easy to work with files
  - Easy to work with strings
  - Great for prototyping. No compilation
- Disadvantages of shell scripts
  - Slow
  - Not well suited for algorithms & data structures

The C Shell

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

The Bourne Shell

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

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

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
```

- We will see others

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>file2`**
  - send standard output to file1
  - send standard error to file2

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
- **format**

```
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 steinbrenner@yankees.com <<EOT
   You guys really blew it in
   yesterday.  Good luck tomorrow.
   Yours,
   $USER
   EOT
```

Shell Variables

- Write

`name=value`

- Read: **`$var`**

- Turn local variable into environment:

`export variable`

Variable Example

```
#!/bin/sh

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

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 login name
$SHELL	Absolute pathname of login shell
$TERM	Type of your terminal
$PS1	Prompt

Parameters

- A parameter is one of the following:
  - A variable
  - A *positional parameter*, starting at 1 (next slide)
  - A *special* parameter
- To get the value of a parameter: **`${param}`**
  - Can be part of a word (**`abc${foo}def`**)
  - Works in double quotes
- The **`{ }`** can be omitted for simple variables, special parameters, and single digit positional parameters.

Positional Parameters

- The arguments to a shell script
  - `$1, $2, $3 …`
- 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**
- Parameter 0 is the name of the shell or the shell script.

Example with Parameters

```
#!/bin/sh

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

```
$ countlines ing /usr/dict/words
3277
```

Special Parameters

- `$#` Number of positional parameters
- `$-` 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
- `"$@"` All arguments on command line individually quoted `"$1" "$2" ...`

Command Substitution

- Used to turn the output of a command into a string
- Used to create arguments or variables
- Command is placed with grave accents ` ` to capture the output of command

```
$ date
Wed Sep 25 14:40:56 EDT 2001
$ NOW=`date`

$ sed "s/oldtext/`ls | head -1`/g"

$ PATH=`myscript`:$PATH
$ grep `generate_regexp` myfile.c
```

File name expansion

- Wildcards (patterns)

* matches any string of characters

? matches any single character

[**list**] matches any character in **list**

[lower-upper] matches any character in range **lower-upper** inclusive

[!list] matches any character not in list

File Expansion

- If multiple matches, all are returned and treated as separate arguments:

```
$ /bin/ls
file1 file2
$ cat file1
a
$ cat file2
b
$ cat file*
a
b
```

- Handled by the shell (*exec never sees the wildcards*)
  - argv[0]: /bin/cat
  - argv[1]: file1
  - argv[2]: file2

NOT

  - argv[0]: /bin/cat
  - argv[1]: file*

Compound Commands

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

Boolean Operators

- Exit value of a program (**exit** system call) 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

```
$ ls bad_file > /dev/null && date
$ ls bad_file > /dev/null || date
Wed Sep 26 07:43:23 2001
```

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`** exists that does most common expressions
  - String compare
  - Numeric comparison
  - Check file properties
- **`/bin/[`** is linked to **`/bin/test`** for syntactic sugar
- Good example UNIX tools working together

Examples

```
if test "$USER" = "kornj"
then
      echo "I hate you"
else
      echo "I like you"
fi
```

```
if [ -f /tmp/stuff ] && [ `wc -l < /tmp/stuff` -gt 10 ]

then
      echo "The file has more than 10 lines in it"
else
      echo "The file is nonexistent or small"
fi
```

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

- **File tests**

`-r file`	File exists and is readable
`-w file`	File exists and is writable
`-f file`	File is regular file
`-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

Control Structures Summary

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