Advanced Unix Tools	**Lecture 2**
CS214 Spring 2002	**Wednesday, February 20, 2002**

Bash has many features that makes it an attractive interactive environment. We will not talk about those in this course. Rather, we will dive into the scripting aspects. In this lecture, we will quickly go over the basics of *bash* that will be relevant when we discuss scripting, things like handling exit codes, metacharacters, variables, quoting, and so on.

Commands

Recall that every command[1] returns an exit code to its parent process (the process that invoked it in the first place, typically the shell), a number between 0 and 255. A value of 0 indicates success (however success is defined for the command at hand), while a value greater than 0 represents failure. Every command may define its own meaning for the failure code (i.e., what does it mean when it returns 1, or 2, or 3). Some error codes are reserved for *bash*. For example, if a command returns an exit code of 127, it means that in fact *bash* was not able to execute the command because it couldn't find it, maybe because of an error in syntax. Similarly, exit codes greater than 127 indicate that the command was interrupted due to the arrival of a signal. (We will not cover signals in this course.)

Commands can be composed in different ways on the command line. The form *cmd1 ; cmd2* executes *cmd1* and then *cmd2*. The exit code returned is the exit code of *cmd2*. The form { *cmd1 ; cmd2 ;* } (the trailing ; is required) is similar to the previous case, except that this form allows redirection to apply to all commands in the list at once. The form *(cmd1 ; cmd2)* is similar to the previous case, except that the commands are executed in a subshell. Finally, conditional execution is expressed in two different ways. The form *cmd1 && cmd2* executes *cmd1*, and if it succeeds (i.e., it returns exit code 0), then executes *cmd2*. Alternatively, *cmd1* || *cmd2* executes *cmd1*, and if it fails (i.e. returns exit code > 0), then executes *cmd2*.

Shell expansions

The shell performs quite a bit of work on any command line it executes, whether it is typed at the prompt or read from a script. The following steps are performed in order, and I will describe most of them in the remainder of the lecture.

[1] I will use the term "command" to stand for either commands builtin to the shell, programs executed from the filesystem, aliases, etc.

1. brace expansion
2. tilde expansion
3. parameter expansion
4. variable substitution
5. command substitution
6. arithmetic substitution
7. word splitting
8. pathname expansion

For sake of discussion, we will call a sequence of characters separated by spaces a word.

Brace expansion is the process of expanding every word containing a brace expression, of the form {*w1,w2,w2*}, where each *w1, w2, w3* are words (without any space), into a sequence of words where the brace expression is replaced by *w1, w2, w3*, respectively. For example, *abc*{*de,fg,hi*} expands into the sequence of words *abcde abcfg abchi*.

Tilde expansion expands every ∼ into the path to your home directory. The form ∼*name* expands into the path to the home directory of user *name*.

Variable substitution and parameter expansion substitute the value of variables. Variable substitution replaces every expression *$var* by the value of variable *var*. Parameter expansion is a kind of conditional substitution. There are many variations of parameter expansion. The most common one is to replace every expression of the form *$*{*var:-word*} either by the value of variable *var* if *var* is set and non-null, or by *word* if *var* does not exist, or is null-valued.

Command substitution replaces every expression of the form *$(cmd)*, where *cmd* is a command, by the output of the execution of *cmd*. Hence, *$(pwd)* is replaced by the output of *pwd*, that is, the current working directory.

Arithmetic substitution allows you to perform numerical computations in the shell, instead of using a program such as *bc*. We will return to arithmetic expressions in later lectures.

Word splitting is the process of actually splitting the command line into words. Hence, if a substitution occuring earlier in the process substitutes a string with whitespace, for example, *$FOO* where variable *FOO* has value *some word*, then the value will be split into words.

Finally, pathname expansion is the process of replacing paths containing wildcard characters (such as * and *?*) by the sequence of paths that match the pattern. Recall that * matches one of more character, *?* matches exactly one character, *[abc]* matches any character between the brackets (here, *a*, *b*, *c*), and *[!abc]* matches any character not in the brackets. (This process is also known as globbing.)

Sometimes, we want to disable some aspects of this automatic expansion by the shell. By and large, this process is known as quoting. Quoting can take many forms. The simplest form is simply to escape the special characters that have meaning to the shell. For instance, you may want to use the *$* character without having it interpreted as variable, parameter or command substitution. Similarly for the *&*, *;*, *?*, ***, *[*, *]*, {, } characters. To use such a special character as a literal character, you precede it with a backslash. (This is called escaping a character.) Since a backslash is itself a special character, if you want a literal backslash, you need to escape it as well.

Two alternate forms of quoting exist. The form *'text'* disables any form of expansion between the single quotes, including word splitting. Hence, a line *cmd 'word1 word2'* will split the line into *cmd* and *word1 word2*. The form *"text"* disables expansion between the double quotes, except for variable and command substitution. Hence, *"*$foo"* will expand to **bar* if variable *foo* has value *bar*. As with single quotes, double quotes also disables word splitting.

Redirection

Recall that redirection allows you to redirect the input to a command from a file (using <), or the output to a file (using >, >>, depending on whether we want to overwrite the file or append to it). A special form of redirection is useful, called a “here document”. It allows you to redirect input not from a file but from explictly provided data. The general form of this form of redirection is as follows:

```
cmd << FLAG
line1
line2
line3
line4
FLAG
```

This executes *cmd*, feeding it *line1*, *line2*, *line3* and *line4* as input. The *FLAG* after << indicates until where to read the input; the first line containing *FLAG* by itself is the end of the input to feed to the command. The terminating expression need not be *FLAG*. It can be anything. Generally, it will be something that does not appear in the text to feed to the command. Note that expansion is performed on the lines to pass to the command. Expansion is *not* performed on *FLAG*.

Here's a concrete example, to mail a piece of text to *joe*:

```
mail -s "value of path var" joe << XYZ
```

Joe, here’s the value of my
PATH variable: $PATH
XYZ