Advanced Unix Tools	**Lecture 1**
CS214 Spring 2004	**Monday, February 23, 2004**

I will assume you are familiar with the CS114 material:

- filesystem notions
- permissions
- text editing
- regular expressions
- job control
- I/O redirection
- shell ideas: variables, scripts

I also assume that you are familiar with the basic ideas of structured programming (conditionals, while loops, for loops, recursion, etc.)

Shells

Recall that on a Unix system, the shell is the basic interface between the user and the operating system. More specifically, the shell has the following "responsabilities":

1. read and parse command line,
2. evaluate special characters,
3. setup pipes, redirections, and background processing,
4. find and setup programs for execution.

The history of shells is tied with that of Unix itself. Originally (I'm talking in the early 70s), there were two main "strains" of Unix: one from AT&T, and one from Berkeley. AT&T Unix came with the Bourne shell (or *sh*, due to Stephen Bourne), a minimalistic shell inspired by Algol, and much more aimed at scripting than at easing the interactive experience. At Berkeley, Bill Joy and others developed the C shell (or *csh*), inspired by the C programming language, and with

interactivity in mind: it supported history mechanisms and job control. In 1986, David Korn developed the Korn shell (or *ksh*), an extension of *sh* that included much of the *csh* features, and more beyond. Finally, the Bourne Again shell (or *bash*) was developed, an extension of *sh*, inspired by *ksh*. It sported history, command-line edition, filename completion, aliases, arrays, and many programming features. It remains *sh*-compatible. Alternatively, in the *csh* world, the TC shell (or *tcsh*) is a recent shell extending *csh* and supporting much of the same features as *bash*, albeit with a different syntax. In this course, we will focus on *bash*, the Bourne Again shell. Why? It is a superset of *sh*, the minimalist shell, is has a slightly easier syntax, it is the default shell on many Linux systems, and finally, why not?

Commands

A general command in *bash* takes the form *cmd arg1 arg2 ... argn* where *cmd* is the command name (or path to an application to execute), and *arg1*–*argn* are the arguments of the command. (I will typically elide the arguments when unnecessary to my examples.)

Many commands read their input from the standard input device (stdin), and send their output to standard output device (stdout). The exceptions are applications that work with interactively specified files, such as text editors, image editors, and so on. By default, the standard input and output device is the terminal, generally the window through which you interact with the Unix machine, and which is running the shell that you are using.

The output of a command can be redirected so that instead of sending it to the terminal, it can be sent to an arbitrary file. The command form *cmd* > *file* executes command *cmd*, but instead of sending output to stdout, it sends it to the file *file*. If this file exists, it is overwritten. If it does not, it is created. As an alternative to overwriting a file, it is possible to append the output to the end of an existing file, using the form *cmd* >> *file*. Again, in this case, if the file does not exist, it is created.

The input of a command can similarly be redirected, so that input is read from a file rather than from the terminal. The command form *cmd* < *file* executes command *cmd*, but instead of reading its input from stdin, it reads it from the file *file*.

We can also combine the notion of output and input redirection, so that the output of a command is fed directly as the input of another command. This is called piping (pipelining), and very common under Unix. The command form *cmd1* | *cmd2* executes command *cmd1*, and sends its output not to the terminal or to a file but rather directly to the command *cmd2*, which is executed using *cmd1*'s output as input. Pipes generalize to more than two commands and can combine with redirection, as in *cmd1* | *cmd2* | *cmd3* | *cmd4* > *filename*.

When a command exists, not only does it perform some input and output (maybe), but it also reports to the shell a status in the form of an *exit code*, a number typically between 0 and 255. An

exit code of 0 means that the command succeeded, while a code different than 0 means that the command failed in some way. The man pages for the command will tell you precisely what exit codes can be returned by the command. For example, `grep` will return an exit code of 0 if it has found at least one match, it will return an exit code of 1 if it not found any matches, and it will return an exit code of 2 if there has been an error (for example, you grep on a file that does not exist).

Shell Scripts

A script is just a "program" made up of shell comamnds. A shell script is a text file with the following characteristics:

- its first line is *#!/bin/bash*, indicating the shell to use to execute the script,
- it has execute permission.

(It is also possible to execute the script by invoking *bash* on the script, as in *bash scriptfile*.) A script is executed in a subshell.

To exit a script with a specific exit code *n*, use the command *exit n*.