CS 114 – Fall 2004

Lecture 5 – Monday, October 5, 2004

I/O redirection

In Unix, a job normally has at least 3 I/O channels it can use to communicate:

- `stdin` – the standard input channel, where it reads its input
- `stdout` – the standard output channel, where it writes its output
- `stderr` – the standard error channel, where it writes its error messages

Most programs print their output to `stdout` and error messages to `stderr` (some screen-oriented programs such as Emacs, pine, or X Windows programs are an exception). Many programs that normally operate on files would operate on `stdin` when no file argument is given. Other programs would allow users to specify `-` as a filename to specify reading from `stdin` or writing to `stdout`. The man pages for a command or program will tell you whether and how it can read from `stdin` or write to `stdout`.

By default, all three channels point to the console (the terminal). However, any of them can be redirected. To redirect `stdin` from a file, you append `<` *`file`* to the command. This will force the command to read the input it reads from `stdin` from the specified file.

To redirect `stdout` to a file, you append `>` *`file`* to the command. This will send all the output that the program writes to `stdout` to the specified file. If the file already exists, it may or may not be overwritten, depending on your shell. You can prevent the file from being overwritten, you can add the command `set noclobber` to your shell rc file.

To append to the end of a file, you can redirect using `>>` *`file`* instead.

You can take a command and connects its output channel to the input channel of another command. This is called a *pipe*. To do so, you put a `|` between the commands, such as:

```
% command1 | command2
```

It is possible to do many redirections at the same time:

```
command1 < infile | command2 | command3 | command4 > outfile
```

This is often called a *pipeline*. The following programs are useful on their own, but are also interesting when used in a pipeline:

- `fgrep string file ...` searches for the string `string` in `file` If no file is specified, it reads from `stdin`.
- `wc file1 file2 ...` prints file sizes (and total), including character, word and line counts.
- `sort` reads a file from `stdin` and outputs to `stdout` the lines sorted in alphabetical order (this can be changed; see the man pages).

As an example of the above, consider searching a big file `f` for text `hello` via `fgrep`. You can sort the resulting lines using the following pipeline:

```
fgrep hello f | sort
```

or similarly count the number of lines (and words, and characters) in the matching lines, using:

```
fgrep hello f | wc
```

Job control

When you run a program from a shell, this running program is called a *job*. So far we've only been running one program at a time, but it is possible and useful to run multiple programs at once. For example, a job may be a very long running program that can work in the background while you do other things. At times you may be required to interrupt a job to do something else, knowing that you'll want to return to the original job later.

A job can be in one of three states:

- *running in the foreground* – the job is currently executing and can read from the console
- *running in the background* – the job is currently executing, but cannot read from the console (although it can write to it). If a job running in the background tries to read from the console, it will be suspended.
- *suspended* – the job is interrupted but can be resumed

By default, when you execute a program from the shell command line, it runs in the foreground. To *suspend* a program, you (typically) press `Ctrl-Z` (Also typically, if you press `Ctrl-C`, it will quit the program.) When you suspend a job, you get an indication that it is suspended. To get a list of all jobs you are running (or are suspended), you can use the command `jobs`, which outputs something like this:

```
$ jobs
[1]+ Stopped vi lec5.html
[2]- Running xterm &
```

For each job, `jobs` returns its *job number*, and the command that initiated the job. In the above example, there are two jobs, one running in the background and one suspended. Note that `jobs` doesn't show any jobs running in the foreground since `jobs` itself is running in the foreground.

To resume a suspended job, use the command `fg`. This will resume the current job (indicated by the `+` in `jobs`). You can resume a specific job by running `fg %`*`job-number`*. Do not forget the `%`.

You can also resume a suspended job, with the `bg` command. This runs the job in the *background*.

You can start a job directly in the background by invoking the command and adding a `&` at the end. For example:

```
$ chmod -R go-rwx ~&
[2] 3270
$
```

will run `chmod` recursively on your home directory in the background. Note that the job number is printed along with another number, called the *process id*. When a job is started in the background, you are immediately returned to the prompt.

You can usually terminate a program running in the foreground by hitting `Ctrl-C` To terminate a suspended program or one running in the background, you can use the command `kill %`*`job-number`*. This will attempt to kill the job nicely (i.e., it will allow the job to perform "last rites"). This doesn't always work. To kill a job "with extreme prejudice", you can use `kill -9 %`*`job-number`*.

Every job in Unix corresponds to one or more *processes*. Processes are lower level than jobs and are the unit of execution in Unix. Some processes correspond to user jobs. Others are started by the system itself. To see the currently running processes use the command `ps`. The output of `ps` looks something like this:

```
$ ps   PID  TT  STAT       TIME COMMAND
 226  p1  S+     0:00.28 -bash
```

```
3261  p1   S       0:01.96 vi lecture5
 734  p2   S+      0:00.54 -bash
 840  p3   S+      0:00.10 -bash
 944  p4   S+      0:00.34 -bash
3789 std   S       0:00.13 -bash
3828 std   S       0:03.06 vi lec5.html
$
```

The first column displays the process id. The last column displays the name of the program that's running (`-bash` indicates a `bash` login shell). `ps` can take command-line options. The options differ depending on which version of Unix you are running.

Command	**on Linux, Mac OS X**	**on Solaris**
default	`ps`	`ps`
list all processes in the current terminal	`ps a`	`ps -t` *`terminal`* (use the `tty` command to get the current terminal)
list all processes of the current user	`ps u`	`ps -u` *`username`*
list all processes running in any terminal	`ps a`	`ps`
list all processes on the system	`ps ax`	`ps -e, ps -A`
wide format	`ps w, ps ww`	`ps -f, ps -l`