Jobs

In Unix, every time you execute a program, this is called a *job*. Up until now, there was only one job at a time executing, namely the command or program you happen to call from the command line. However, it is useful to be able to have many jobs 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, but knowing that you'll want to return to the original job and pick up where you left off.

A job can be in one of three states:

- *suspended*, meaning it is interrupted, but can be resumed

- *running in the foreground*, meaning it is currently executing and it can read and write to the console

- *running in the background*, meaning it is running but cannot read and write from the console.

By default, if you just execute a command or a program as usual, it runs in the foreground. To *suspend* a program, you (typically) press CONTROL-z. (Also typically, if you press CONTROL-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 the jobs you are running (or are suspended), you can use the command `jobs`, which outputs something like this:

```
babbage% jobs
[1]  - Suspended (tty output)      emacs -nw
[2]  + Suspended (tty input)       cat
[3]    Running                     xterm
```

For each job, `jobs` returns its *job number*, its status, and the command that initiated the job. In the above example, there are three jobs, two suspended, and one running in the background (since `jobs` is running in the foreground!) Job numbers are assigned by Unix, in increasing order.

How do you resume a suspended job? You use the command `fg %job-number`. Do not forget the %. This will resume the given job and put it in the foreground. (If you simply put `fg`, the current job, for instance the last job you suspended, is resumed.)

If you use the command bg `%job-number` instead, the given job is resumed but in the background. (Again, you can omit the job number.) I said earlier that a job running in the background cannot read or write to the console. What happens if the program you want to resume needs to read or write to the console? Unix will not allow you to put it in the background, and will instead suspend it automatically if you attempt a bg. In the sample output for jobs above, the term ttyoutput and ttyinput respectively indicate that the job require console output or console input, and therefore cannot be put in the background.

You can also start a job directly in the background, by invoking the command and adding a & at the end of the line. We haven't seen too many examples of programs that can be run in the background, but say that you have a long job such as doing a chmod recursively in a huge directory. You can do this in the background by invoking, say, chmod -R og-rwx `directory` &.

You can also kill a job that is either suspended or running in the background. The command kill `%job-number` will attempt to nicely kill the job (i.e. will allow the job to perform "last-rites"). This doesn't always work. To kill a job "with extreme prejudice", you can always go for the kill -9 `%job-number` options.

(To every job in Unix corresponds one or more processes. Processes are lower level, and are actually the unit of execution that Unix understands. Some processes correspond to user jobs, some processes are started by the system itself to handle its own processing. To see the processes you are currently running, try ps. If you want to see all the processes associated with an actual user, try ps -a. If you want to see all the processes in the machine, try ps -e. That gives you an idea of the amount of work an operating system performs.)

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, for example, grep. 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 is (most of the time) overwritten. This "most of the time" depends on the shell you are using. We will cover shells next week. 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:

- `wc file1 file2 ...` prints file sizes (and total), including character, word and line counts. If no file is specified, it reads from `stdin`.

- `sort` reads a file from `stdin` and outputs to `stdout` the lines sorted in alphabetical order (that 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`.