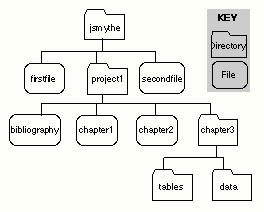
**Chapter 11**

**Linking Files and Background/Foreground Processes**

**1. Linking files**

The ln command (short for link) lets we give multiple names to a single file. This is useful when we want to get at a file quickly from within different directories. Assume that our directory structure looks like this:



We may need to refer frequently to a file called table1 in the tables subdirectory when we are in our home directory (jsmythe). Rather than typing the whole pathname

**project1/chapter3/tables/table1**

we could link the file to our home directory. If we are in the home directory, the command to do this is:

**$ ln project1/chapter3/tables/table1 mytable1**

To create the link when we are in the tables directory, the command would have been:

**$ ln table1 ~/mytable1**

After issuing either of these commands, an ls command in our home directory would show an entry for mytable1. The long format of the same command would show 2 links for the file mytable1:

**$ ls -l**

**-rw------- 2 jsmythe 6 Jul 4 14:23 mytable1**

A long format listing of the file table1 would also show 2 links. What if a file called mytable1 had already existed in our home directory? Unix would let we know that a file by that name exists, and would not make the link.

The effect of linking is that the file now has two names. We may call up the file by either name. Creating a link does not change the ownership, group, or permissions for a file. The inode number of two files are same.

**1.1 Removing links**

Links are removed using the rm command. To continue the example above, the command

**$ rm mytable1**

Removes one link to the file table1 by removing the file mytable1. The file table1 itself, and its contents, still exists. Only when all the links to the file have been removed will the file itself be erased.

**1.2 Symbolic links**

The links described in the sections above are "hard" links. In effect, making a hard link creates a standard directory entry just like the one made when the file was created. Hard links have certain limitations. Hard links cannot be made to a directory, only to files, and hard links cannot be made across file systems and disk partitions.

There is another kind of link, called a symbolic link. Symbolic links can span file systems, and can be made for directories.

In the figure above, assume that we want to make a symbolic link from our home directory (jsmythe) to the directory called chapter3. To create the symbolic link, we would move to our home directory and give the command:

**$ ln -s project1/chapter3 linkdir**

The -s option means that a symbolic link is created (instead of the default hard link). This command creates a symbolic link called linkdir which points to the directory called chapter3.

When we list the contents of linkdir

**$ ls linkdir**

**$ data tables**

we see a listing of the contents (data and tables)

We can learn more about the new directory by using the long format and directory options with the ls command:

**$ ls -ld linkdir**

**l--------- 1 staff 7 Jun 11 13:27 linkdir -> project1/chapter3**

Symbolic links are shown with an arrow (->) in the name column at the right.

**1.3 Removing symbolic links**

Use rm to remove a symbolic link, for example

**$ rm linkdir**

This removes the link only, not the file or directory it was pointing to. If the file or directory is removed but the link remains, the link will no longer work.

**2. Unix Processes**

A process is an instance of running a program. If, for example, three people are running the same program simultaneously, there are three processes there, not just one. In fact, we might have more than one process running even with only person executing the program, because (we will see later) the program can ``split into two,'' making two processes out of one.

Keep in mind that all Unix commands, e.g. cc and mail, are programs, and thus contribute processes to the system when they are running. If 10 users are running mail right now, that will be 10 processes. At any given time, a typical Unix system will have many active processes, some of which were set up when the machine was first powered up.

Every time we issue a command, Unix starts a new process, and suspends the current process (the C-shell) until the new process completes (except in the case of background processes, to be discussed later).

Unix identifies every process by a Process Identification Number (pid) which is assigned when the process is initiated. When we want to perform an operation on a process, we usually refer to it by its pid.

Unix is a timesharing system, which means that the processes take turns running. Each turn is a called a timeslice; on most systems this is set at much less than one second. The reason this turns-taking approach is used is fairness: We don't want a 2-second job to have to wait for a 5-hour job to finish, which is what would happen if a job had the CPU to itself until it completed.1

**2.1 Determining Information about Current Processes**

The `ps -x' command will list all our currently-running jobs. An example is:

**$ PID TT STAT TIME COMMAND**

**6799 co IW 0:01 -csh[rich] (csh)**

**6823 co IW 0:00 /bin/sh /usr/bin/X11/startx**

**6829 co IW 0:00 xinit /usr/lib/X11/xinit/xinitrc --**

**6830 co S 0:12 X :0**

**6836 co I 0:01 twm**

**6837 co I 0:01 xclock -geometry 50x50-1+1**

**6841 p0 I 0:01 -sh[rich on xterm] (csh)**

**6840 p1 I 0:01 -sh[rich on xterm] (csh)**

**6847 p2 R 0:00 ps -x**

The meaning of the column titles is as follows:

|  |  |
| --- | --- |
| PID | process identification number |
| TT | controlling terminal of the process |
| STAT | state of the job |
| TIME | amount of CPU time the process has acquired so far |
| COMMAND | name of the command that issued the process |

The TT information gives terminal names, which we can see by typing the who command. E.g. we see p2 in the TT column above, which is the terminal listed as ttyp2 in the who command.

The state of the job is given by a sequence of four letters, for example, `RWNA'. The first of these four is typically one of the following:

first letter runnability of the process R runnable process T stopped process S process sleeping for less than about 20 seconds we processes that are idle (sleeping longer than about 20 seconds) A state-R process is runnable, i.e. it is be able to make use of a turn given to it, and is waiting for one. We can put a process in state T, i.e. stop the process, by typing control-z. Suppose, for example, that we are using ftp to get some files from some archive site, and we notice a file there called README. we can use the ftp `get' command to get the README file, and then type C-z. This will stop (suspend) the ftp process, and get me back to the C-shell. At that point we can read the README file, say using more, and then reactivate the ftp process, by typing `fg' to the shell.

A typical example of an S/I process is one that is waiting for user input. If we are using the emacs editor, for example, the process will go to state S when it is waiting for me to type something; if we take more than 20 seconds to decide what to type, the process will be in state I.

**2.2 Foreground/Background Processes**

Suppose we want to execute a command but do not want to wait for its completion, i.e. we want to be able to issue other commands in the mean time. We can do this by specifying that the command be executed in the background.

There are two ways to do this. The first is to specify that it be a background process when we submit it, which we can do by appending an ampersand (`&') to the end of the command. For example, suppose we have a very large program, which will take a long time to compile. We could give the command

**$ cc bigprog.c &**

Which will execute the compilation while allowing me to submit other commands for execution while the compile is running. The C-shell will let me know what the pid is for this background process (so that we can later track it using ps, or kill it), but will also give me my regular prompt, inviting me to submit new commands while the other one is running.

But what about the compiler error messages? We hope we don't have any :-) but if we do have some, we don't want them to be interspersed with the output of other commands we are running while the compile is executing. To avoid this, we redirect the error messages:

**$ cc bigprog.c >& errorlist &**

All error messages will now be sent to the file `errorlist', which we can view later.

Another good example is when we start a window during a X session. We would like to start the window from an existing window, but we still want to be able to use the original window. W e execute the command

**$ xterm &**

This will start a new window, and allow us to keep using the current window.

The other way to put a job in the background is to stop it, using C-z as described earlier, and then use another command, bg, to move the process to the background.

For example, suppose we started our long-running compile,

**$ cc bigprog.c**

but we forget to append the ampersand. We can type control-z to suspend/stop the job, and then type `bg' to resume the job in the background, allowing us to submit other commands while the compilation takes place. Unix will tell us when the background job has completed, with a statement like

**[1] Done cc bigprog.c**

By the way, if we log out, whatever background processes we have running at the time will not be killed; they will continue to run.

**2.3 Terminating a Process**

We can terminate a process by using the kill command. We simply find its pid (say by using ps), and then type

**$ kill -9 pid**

If the process is in the foreground, though, the easiest way to kill it is to simply type control-C.

**Notes:**

* The basic mechanism for setting up the turns is as follows. The machine will have a piece of hardware which sends electrical signals to the CPU at periodic intervals. These signals force the CPU to stop the program it is running, and jump to another program, which will be the operating system program (OS). The OS can then determine whether the current program's timeslice is finished, and if so, then give a turn to another program, by jumping to that program. Note the interaction of hardware (the electrical signals, and the CPU's reaction to them) and software (the OS) here.
* Keep in mind, though, that there is ``no free lunch'' here. The more processes on the machine, the longer it is between turns for each process, so overall response time goes down.
* Though a much better solution to the problem is to use emacs, since the error messages will automatically be placed into a special buffer.

**2.4 Processes and Jobs**

A process is an executing program identified by a unique PID (process identifier). To see information about wer processes, with their associated PID and status, type

**$ ps**

A process may be in the foreground, in the background, or be suspended. In general the shell does not return the UNIX prompt until the current process has finished executing. Some processes take a long time to run and hold up the terminal. Backgrounding a long process has the effect that the UNIX prompt is returned immediately, and other tasks can be carried out while the original process continues executing.

**2.4.1 Running background processes**

To background a process, type an & at the end of the command line. For example, the command sleep waits a given number of seconds before continuing. Type

**$ sleep 10**

This will wait 10 seconds before returning the command prompt %. Until the command prompt is returned, we can do nothing except wait.

To run sleep in the background, type

**$ sleep 10 &**

**[1] 6259**

The & runs the job in the background and returns the prompt straight away, allowing we do run other programs while waiting for that one to finish. The first line in the above example is typed in by the user; the next line, indicating job number and PID, is returned by the machine. The user is be notified of a job number (numbered from 1) enclosed in square brackets, together with a PID and is notified when a background process is finished. Backgrounding is useful for jobs which will take a long time to complete.

**2.4.2 Backgrounding a current foreground process**

At the prompt, type

**$ sleep 100**

We can suspend the process running in the foreground by holding down the [control] key and typing [z] (written as ^Z) Then to put it in the background, type

**$ bg**

Note: do not background programs that require user interaction e.g. pine

**2.4.3 Listing suspended and background processes**

When a process is running, backgrounded or suspended, it will be entered onto a list along with a job number. To examine this list, type

**$ jobs**

An example of a job list could be

**[1] Suspended sleep 100**

**[2] Running netscape**

**[3] Running nedit**

To restart (foreground) a suspended processes, type

**$ fg %jobnumber**

For example, to restart sleep 100, type

**$ fg %1**

Typing fg with no job number foregrounds the last suspended process.

**3. Killing a process**

kill (terminate or signal a process), It is sometimes necessary to kill a process (for example, when an executing program is in an infinite loop) To kill a job running in the foreground, type ^C (control c). For example, run

**$ sleep 100**

**^C**

To kill a suspended or background process, type

**$ kill %jobnumber**

For example, run

**$ sleep 100 &**

**$ jobs**

If it is job number 4, type

**$ kill %4**

To check whether this has worked, examine the job list again to see if the process has been removed.

**3.1 ps (process status)**

Alternatively, processes can be killed by finding their process numbers (PIDs) and using kill PID\_number

**$ sleep 100 &**

**$ ps**

**PID TT S TIME COMMAND**

**20077 pts/5 S 0:05 sleep 100**

**21563 pts/5 T 0:00 netscape**

To kill off the process sleep 100, type

**$ kill 20077**

and then type ps again to see if it has been removed from the list.

If a process refuses to be killed, uses the -9 option, i.e. type

**$ kill -9 20077**

Note: It is not possible to kill off other users' processes !!!