Unix Systems: Some more useful commands

Bob Dowling

rjd4@cam.ac.uk

20 July 2006

This is a one afternoon course given by the University Computing Service as part of its "Unix Systems" series of courses. It presumes that the reader has either attended the "Unix Systems: Introduction" course or has equivalent experience. The introductory course teaches the use of the commands "`ls`", "`mv`", "`mkdir`" etc. This course introduces a further set of commands.

Table of Contents

The locate command

The `locate` program is used to find files of a certain name on the system. Suppose that on the system — somewhere — you knew there was a file called `catalog.xml` but you didn't know where. Running the command “`locate catalog.xml`” will locate all the files on the system with “`catalog.xml`” in their names:

```
$ locate catalog.xml
/etc/xml/suse-catalog.xml
/opt/kde3/share/apps/ksgmltools2/docbook/xml-dtd-4.2/catalog.xml
/opt/netbeans-4.1/ide5/config/Modules/org-netbeans-modules-xml-catalog.xml
/opt/netbeans-4.1/ide5/update_tracking/org-netbeans-modules-xml-catalog.xml
/usr/matlab/sys/namespace/docbook/v4/dtd/catalog.xml
/usr/matlab/sys/namespace/mcode/v1/catalog.xml
/usr/share/docbook2X/xslt/catalog.xml
/usr/share/sgml/docbook/dtd/4.2/catalog.xml
/usr/share/sgml/docbook/dtd/4.3/catalog.xml
/usr/share/sgml/docbook/dtd/4.4/catalog.xml
/usr/share/sgml/tei/dtd/4.0/catalog.xml
/usr/share/sgml/tei/dtd/4.0/teicatalog.xml
/usr/share/xml/docbook/custom/website/2.5.0/catalog.xml
/usr/share/xml/docbook/custom/website/2.5.0/example/catalog.xml
/usr/share/xml/docbook/schema/dtd/4.2/catalog.xml
/usr/share/xml/docbook/schema/dtd/4.3/catalog.xml
/usr/share/xml/docbook/schema/dtd/4.4/catalog.xml
/usr/share/xml/entities/xmlcharent/0.3/catalog.xml

$
```

You may have noticed that your hard drives did not spring into life to locate these files. The `locate` program doesn't scan the system each time you make a query. Instead, once a night a single sweep of the system is performed[1] which creates a database of filenames. It is this database that is searched when you issue the `locate` command.

By default, `locate` will list any file or directory which has its search string within it. You can use “`/`” within the search string to identify directories. The program can generate huge amounts of output and its output is often piped into `grep` for more refined searching.

Your own locate database

There is a downside to rebuilding the database each night. You aren't logged in, so your home directory isn't mounted and so it isn't indexed. You can build your own `locate` database for your own personal use and then tell `locate` to use it.

To build the database in the first place, use the “`updatedb`” command. This will need a couple of options to tell it to create a database in your home directory (rather than in the system location) and reading your home directory (and not the system files):

```
$ cd

$ updatedb --output=locate.db --localpaths=${HOME}

$
```

This will create a file in your home directory called `locate.db`. It can be called anything you want. (The “`${HOME}`” term is a way to write “your home directory” that the system will convert into your real home directory.)

Next, we have to tell `locate` to use it. The locate command has a “`--database`” option to tell

1 which *does* rattle the hard drive

it what database to use:

```
$ locate syllabics.pdf

$ locate --database=${HOME}/locate.db syllabics.pdf
/home/rjd4/deeply/hidden/directories/syllabics.pdf

$
```

Finally, you can set an environment variable to list the set of databases you want `locate` to search. The variable is called `LOCATE_PATH` and is a colon-delimited list of filenames. The system database is in `/var/lib/locatedb` so if you run the command

```
$ export LOCATE_PATH=/var/lib/locatedb:${HOME}/locate.db

$
```

then the `locate` command will find your files as well as the system ones.

```
$ locate syllabics.pdf
/home/rjd4/syllabics.pdf

$ locate resolv.conf
/etc/resolv.conf
/usr/share/man/man5/resolv.conf.5.gz

$
```

If you put the “`export`” line in a file called “`.bashrc`” in your home directory, then it will be run automatically every time you log in.

You will still need to update your database once in a while.

The find command

The `locate` program uses a database to avoid searching the directory hierarchy every time it is called. A program that *does* walk over an entire directory tree (and potentially the whole file system) is `find`. This command runs over a directory tree looking at each file and directory in turn and applies a series of tests to them. The final "test" is typically either "print the file's name" or "execute this command on the file". It is the `find` command that is run once a night to create the database for `locate` to use.

If we run the command "`find . -print`" then, starting with the current directory ("`.`"), find will run through the entire directory tree and print the name of every file and directory it comes across:

```
$ find . -print
.
./.xsession-errors
./.dmrc
./.pwf-linux
./.pwf-linux/release
./.gconfd
 …
./OpenOffice.org1.1/setup
./OpenOffice.org1.1/soffice
./OpenOffice.org1.1/spadmin
$
```

Types of node

A simple test that find offers is on the type of the file. If we run the command "`find . -type d -print`" then find runs through every file and directory under the current working directory and on each one it runs the test "`-type d`" which means "is it a directory?" If the object in question is not a directory then the processing of that node in the filesystem stops immediately and `find` moves on to the next in its list. It is is a directory then `find` moves on to its next test, which is the pseudo-test "`-print`" in this case, so it prints the name of the node. What this does it to find and print the names of all directories at or below the current one.

```
$ find . -type d -print
.
./.pwf-linux
./.gconfd
./.gconfd/lock
./.gconf
./.gconf/desktop
 …
./OpenOffice.org1.1/share/uno_packages
./OpenOffice.org1.1/program
./OpenOffice.org1.1/program/addin
$
```

Names of nodes

A similar test checks the name of a node in the file system against a shell-style regular expression. If we run the command "`find . -name '[A-Z]*' -print`" then any node below the current working directory will be checked to see if its name starts with a capital letter. If it does then it will be printed. We can combine tests to find just the directories below the current working directory that begin with a capital. Note the use of the single quotes around the regular expression. These stop the shell expanding the expression into a list of the matching files in the current working directory and allow it to be passed unexpanded into `find`.

```
$ find . -name '[A-Z]*' -print
./.gnome-desktop/Wastebasket
./.gnome/Gnome
./.gnome/README
 …
./OpenOffice.org1.1/user/registry/data/org/openoffice/ucb/Store.xcu
./OpenOffice.org1.1/user/registry/data/org/openoffice/ucb/Hierarchy.xcu
./OpenOffice.org1.1/THIRDPARTYLICENSEREADME.html

$
```

Case-insensitive searching can be done by using the option “`-iname`” in place of “`-name`”.

Combining tests

Now that we have two distinct real tests we can illustrate combining them. The following test checks every node to see whether it is a directory and, if it is, whether it starts with a letter between A and Z.

```
$ find . -type d -name '[A-Z]*' -print
./.Trash/OpenOffice.org1.1
./.Trash/OpenOffice.org1.1/user/basic/Standard
./.Trash/OpenOffice.org1.1/user/registry/data/org/openoffice/Office
./OpenOffice.org1.1
./OpenOffice.org1.1/user/basic/Standard
./OpenOffice.org1.1/user/registry/data/org/openoffice/Office

$
```

Sizes of files

Another very useful test is to be able to identify files by size and the “-size” option does this. A slightly subtlety is required, though. The option “-size 100k” will match files that are exactly 100KB in size and it's the option “-size +100k” that will find those larger than 100KB, which is probably what was wanted. Finally, the option “-find -100k” will find those smaller than 100KB.

```
$ find . -type f -size +500k -print
./.adobe/AdobeFnt06.lst.pcphxtr01
./.adobe/AdobeFnt06.lst.smaug
./.mozilla/default/25p8sbwm.slt/XUL.mfasl
./.mozilla/firefox/dy8ua6ci.default/XUL.mfasl
./.openoffice/instdb.ins
./.OpenOffice.org/instdb.ins
./chile/sshot1.ps
./chile/sshot2.ps
./chile/sshot3.ps
./DiscMaths.ps
./FortranNag/nag_lib_support.mod

$
```

The “`k`” at the end of the size stands for KB (kilobytes). Similarly you can use “`500M`” for 500MB and “`500G`” for 500GB. If you want to specify an exact size then use “`c`” to mean “characters” as “`b`” is already taken to mean the quite useless “blocks”. A block is a 512-byte unit of measure that is unused outside of hardware and file system design.

Running commands

In addition to just printing a node's name it is also possible to get find to run a command on the matching node with the "`-exec`" option. Its syntax is rather baroque, though. Suppose we want to run "`wc -l`" on every file that ends in "`.html`". The command we need to run is "`find . -type f -name '*.html' -exec wc -l {} \;`".

```
$ find . -type f -name '*.html' -exec wc -l {} \; | more
72 ./.mozilla/firefox/dy8ua6ci.default/bookmarks.html
114 ./.openoffice/LICENSE.html
368 ./.openoffice/README.html
157 ./.OpenOffice.org/LICENSE.html
100 ./.OpenOffice.org/README.html
496 ./.OpenOffice.org/THIRDPARTYLICENSEREADME.html
 …
38 ./PWF-Linux talk/swrules.html
32 ./PWF-Linux talk/wstncost.html
27 ./PWF-Linux talk/wstnfor.html
27 ./PWF-Linux talk/wstnis.html

$
```

The bizarre hieroglyphs after the "`-exec`" demand some explanation. Immediately following the option is the command to be run, with any options it might have itself. Within these options the token "`{}`" is expanded into the full filename that has been matched and the "`\;`" marks the end of the command. Note that the space before the "`\;`" must be there.

Summary of options covered

There are very many other tests that find can run. The `find` manual and information pages ("`man find`" and "`info find`") list them all.

`-iname`	*reg.exp.*	Node's name matches the regular expression ignoring case
`-name`	*reg.exp.*	Node's name matches the regular expression
`-size`	*XX*	Node has size exactly *XX*
	+*XX*	Node is bigger than *XX*
	-*XX*	Node is smaller than *XX*
	*X*c	Size measured in bytes (**c**haracters)
	*X*k	Size measured in **k**ilobytes
	*X*M	Size measured in **M**B
	*X*G	Size measured in **G**B
`-type`	f	Node is a plain **f**ile
	d	Node is a **d**irectory
	l	Node is a symbolic **l**ink
`-print`		Print the node name. (Typically the default action.)
`-exec`	*command*	Run command on the node.
	{}	Replaced by node name.
	\;	Marks the end of the command being passed to `-exec`.

The du command

A very common file to look for is “the big one that's eating up my quota”. We have seen the “-size” option on find to help us here but there are other commands available to us also.

On the subject of subject of file sizes the other common question is “how much space does this directory take up?” Both these questions can be answered with the du command.

The du command (“du” stands for “**d**isc **u**se”) indicates how much space is taken up by a file or, more typically, a directory's contents. To see what it does change directory to /ux/Lessons/FurtherUnix/dudemo and run the du command.

```
$ cd /ux/Lessons/FurtherUnix/dudemo

$ pwd
/ux/Lessons/FurtherUnix/dudemo

$ du
18      ./alpha/alpha
21      ./alpha/beta
18      ./alpha/gamma
19      ./alpha/delta
 …
17      ./alpha/psi
17      ./alpha/omega
443     ./alpha
17      ./beta/alpha
18      ./beta/beta
19      ./beta/gamma
18      ./beta/delta
 …
18      ./omega/psi
18      ./omega/omega
453     ./omega
10643   .

$
```

The command runs through the directory tree quoting each low level directory (alpha/alpha, alpha/beta, alpha/gamma, etc.) first and then giving a total for the mid-level directory (alpha, beta, etc.) and finally the the top-level directory, . (the current working directory).

If we just want to know the intermediate sizes (for example to find the directory tree containing the large file) we can ask for just the summary information for the quoted directory with the “-s” (for **s**ummary) option:

```
$ du -s *
443     alpha
448     beta
436     chi
 …
449     upsilon
438     xi
447     zeta

$
```

Of course we can ask the same question about the top level directory:

```
$ du -s .
11390   .

$
```

The du command is often run in conjunction with the “sort” command to give the largest values first or last:

```
$ du -s * | sort --numeric
433     delta
435     eta
435     omicron
 …
450     kappa
453     omega
1187    lambda

$ du -s * | sort --numeric --reverse
1187    lambda
453     omega
450     kappa
 …
435     omicron
435     eta
433     delta

$
```

We can use this trick to tunnel into a directory tree to find the over-large file. (The “head” function chops off the first few lines of output.)

```
$ du -s * | sort --numeric --reverse | head -3
1187    lambda
453     omega
450     kappa

$ cd lambda/

$ du -s * | sort --numeric --reverse | head -3
767     chi
21      alpha
20      theta

$ cd chi/

$ du -s * | sort --numeric --reverse | head -3
748     psi
1       zeta
1       xi

$ ls -l psi
-rw-r--r--  1 rjd4 rjd4 765432 2006-02-05 16:19 psi

$
```

The ps program

This course assumes you already have some basic knowledge of the `ps` command. The `ps` program has just a few options that you actually use from day to day. It has many other options that you might use once in your life or in the middle of a particular shell script.

The options we will consider can be split into two classes: process selection options, controlling which processes should be reported on, and display formatting options which control how the data about the selected processes should be displayed.

Process selection options

`-e`	Every process
`-U` *user*	Processes owned by *user*.
`-G` *group*	Processes owned by *group*.
`-p` *pid_1,pid_2,pid_3*	Processes with IDs *pid_1*, *pid_2* or *pid_3*.
`-t` *terminal*	Processes running on the given terminal.

```
$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:09 init
    2 ?        00:00:00 migration/0
    3 ?        00:00:00 ksoftirqd/0
 …
10364 ?        00:00:00 sshd
10370 ?        00:00:00 pwfacmd
10400 ?        00:00:00 mount.ncpfs
10424 ?        00:00:00 automount
10435 ?        00:00:00 mount.ncp
10442 pts/8    00:00:00 bash
  952 pts/8    00:00:00 ps

$
```

Note that `ps` can spot itself running as PID 952 in the screenshot above.

```
$ ps -U rjd4
  PID TTY          TIME CMD
10400 ?        00:00:00 mount.ncpfs
10435 ?        00:00:00 mount.ncp
10442 pts/8    00:00:00 bash
10461 ?        00:00:00 mount.ncp
  939 ?        00:00:00 mount.ncp
  963 pts/8    00:00:00 ps

$
```

and as 963 in this one

```
$ ps -p 963,10442
  PID TTY          TIME CMD
10442 pts/8    00:00:00 bash

$
```

and not as 963 in this one which is why only one line is listed when two PIDs were asked for. This third run of `ps` has its own PID.

```
$ ps -t pts/8
  PID TTY          TIME CMD
10442 pts/8    00:00:00 bash
 1137 pts/8    00:00:00 ps

$
```

Finally, we have a listing of all the processes running in this terminal. There is an appendix to these notes explaining the structure of terminal names if you are interested.

Simple column selection options

Note that for each of the rows printed exactly the same set of columns was generated. The quickest way to get lots of data about the processes of interest is to request "full" output with the "`-f`" option. This gives the user, process ID, parent process ID, start time, controlling terminal, CPU time used and command arguments. It is typically more useful than the "long output" option, "`-l`". In theory, the "full" output gives the user all the information he or she could want and the "long" output gives all the system data about the process. There is also an option "`--forest`" that gives the output stylized as a tree to show the parent/child process relationship.

`-f`	"Full" output
`-l`	"Long" output
`--forest`	"Tree" format output

Full output:

```
$ ps -U rjd4 -f

UID        PID  PPID  C STIME TTY          TIME CMD
root     10400     1  0 Jul10 ?        00:00:00 ncpd
root     10435     1  0 Jul10 ?        00:00:00 ncpd
rjd4     10442 10364  0 Jul10 pts/8    00:00:00 -bash
root     10461     1  0 Jul10 ?        00:00:00 ncpd
root       939     1  0 09:46 ?        00:00:00 ncpd
rjd4      1222 10442  0 10:19 pts/8    00:00:00 ps -U rjd4 -f

$
```

Long output:

```
$ ps -U rjd4 -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
5 S     0 10400     1  0  75   0 -   464 -      ?        00:00:00 mount.ncpfs
5 S     0 10435     1  0  75   0 -   481 -      ?        00:00:00 mount.ncp
4 S  2049 10442 10364  0  75   0 -  1107 wait   pts/8    00:00:00 bash
5 S     0 10461     1  0  75   0 -   481 -      ?        00:00:00 mount.ncp
5 S     0   939     1  0  75   0 -   481 -      ?        00:00:00 mount.ncp
0 R  2049  1374 10442  0  76   0 -   673 -      pts/8    00:00:00 ps

$
```

Tree output:

```
$ ps -U rjd4 --forest
  PID TTY          TIME CMD
10442 pts/8    00:00:00 bash
 1385 pts/8    00:00:00  \_ ps
  939 ?        00:00:00 mount.ncp
```

```
10461 ?        00:00:00 mount.ncp
10435 ?        00:00:00 mount.ncp
10400 ?        00:00:00 mount.ncpfs

$
```

Detailed column selection

But if you need particular bits of information, you should consider using the “`-o`” option to specify exactly which output columns you want:

```
$ ps -U rjd4 -o pid,ppid,cmd
  PID  PPID CMD
10400     1 ncpd
10435     1 ncpd
10442 10364 -bash
10461     1 ncpd
939       1 ncpd
1406  10442 ps -U rjd4 -o pid,ppid,cmd

$
```

There are many output options for the “-o” option but the most useful are given here:

`args`	the arguments of the command (including the command)
`cmd`	the command
`pcpu`	percentage of CPU currently used
`time`	CPU time used so far
`stime`	start time
`pmem`	percentage of memory currently used
`rss`	resident set size
`user`	user
`uid`	numeric user ID
`group`	group
`gid`	numeric group ID
`pid`	process ID
`ppid`	parent process ID
`tty`	controlling terminal

A full list is given in the `ps` manual page.

The kill command

So what's the point of identifying running processes? Often it's to find a rogue process that's burning your CPU or eating all your memory so that you can kill it. So that's what we will cover next: killing processes.

To illustrate this we will run the `xclock` program, updating every second:

```
$ xclock -update 1 &
[1] 1521

$
```

Now let's spot it in the `ps` output:

```
$ ps -U rjd4
  PID TTY          TIME CMD
10400 ?        00:00:00 mount.ncpfs
10435 ?        00:00:00 mount.ncp
10442 pts/8    00:00:00 bash
10461 ?        00:00:00 mount.ncp
  939 ?        00:00:00 mount.ncp
 1521 pts/8    00:00:00 xclock
 1526 pts/8    00:00:00 ps

$
```

So it's process number 1521. (Your number will differ, obviously.) Now we will kill it off from the command line:

```
$ kill 1521

$

[1]+ Terminated            xclock -update 1

$
```

The "`kill`" command has lived up to its name; it has killed the `xclock` process. The "terminated" line is the shell notifying you that a backgrounded job has finished and is not the output from kill which produced no output of its own. We can see this most easily by running the `xclock` from one terminal window and the `kill` in another.

1st terminal window	**2nd terminal window**
$ xclock -update 1&	
[1] 1547	
$	
	$ kill 1547
	$
[1]+ Terminated xclock -update 1	
$	

What the `kill` command actually did was to send a "signal" to the `xclock` process. A signal is an asynchronous message, and Unix programs are written to deal with these messages out of the blue. (Actually they tend to follow a set of default behaviours because writing your own signal handler is difficult.)

By default the `kill` program sends a "terminate" signal (also known by its capitalized abbreviation "TERM") to the process. We could equally well have written this:

```
$ kill -TERM 1547
```

The terminate signal is a polite request to a process to drop dead. The process, if it has a handler written for TERM, can put its effects in order, tidy up any files it has only partially written, etc. and then commit suicide. This is what almost all programs do on receiving this signal.

A program can be written to ignore TERM or, if it has gone wrong, might not handle TERM the way it should. In this case we can increase the strength of our signal with the KILL signal:

```
$ xclock -update 1 &
[1] 1611

$ ps -U rjd4
  PID TTY          TIME CMD
10400 ?        00:00:00 mount.ncpfs
10435 ?        00:00:00 mount.ncp
10442 pts/8    00:00:00 bash
10461 ?        00:00:00 mount.ncp
  939 ?        00:00:00 mount.ncp
 1611 pts/8    00:00:00 xclock
 1616 pts/8    00:00:00 ps

$ kill -KILL 1611

$
[1]+  Killed                  xclock -update 1

$
```

The KILL signal cannot be ignored by a process or have its behaviour changed by the program's author. This is the “drop dead now” signal. The process gets no opportunity to put its affairs in order. It just has to die.

If you ever have to kill a process, always start with TERM (the default) and only proceed to KILL if the process hasn't ended in the ten seconds or so following the TERM. If KILL doesn't work then there is nothing you can do. Something has gone wrong at the system level and you can't do anything about it without system administration privileges and even then that may not be sufficient.

Note also that you can't kill processes that don't belong to you:

```
$ ps -U ntp
  PID TTY          TIME CMD
 6305 ?        00:00:00 ntpd

$ kill -TERM 6305
-bash: kill: (6305) - Operation not permitted

$
```

(The `ntpd` is the “network time protocol daemon” and has responsibility for keeping the system clock right.)

There are a few other signals that might be useful so we'll cover them here. Sending interrupt signal (INT) to a process is exactly equivalent to hitting Ctrl+c in the terminal while that process is running in the foreground (i.e. not backgrounded by ending the command with an ampersand):

```
$ xclock -update 1 &
[1] 1648
```

```
$ ps -U rjd4
  PID TTY          TIME CMD
10400 ?        00:00:00 mount.ncpfs
10435 ?        00:00:00 mount.ncp
10442 pts/8    00:00:00 bash
10461 ?        00:00:00 mount.ncp
  939 ?        00:00:00 mount.ncp
 1648 pts/8    00:00:00 xclock
 1653 pts/8    00:00:00 ps
$ kill -INT 1648
$
[1]+  Interrupt               xclock -update 1
$
```

More interestingly there are also signals to pause and restart a process: STOP and CONT ("continue"). Note that the second hand stops moving between the STOP and CONT signals (which is rather hard to show in printed notes).

```
$ xclock -update 1 &
[1] 1716
$ kill -STOP 1716
$
[1]+  Stopped                 xclock -update 1
$
$
$ kill -CONT 1716
$
```

There are plenty more signals but the rest don't really concern us. If you want to see them all run `kill` with the "`-l`" (for "list") option:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1
36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4  39) SIGRTMIN+5
40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8  43) SIGRTMIN+9
44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5
60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2  63) SIGRTMAX-1
64) SIGRTMAX
```

"SIGKILL" is the signal name corresponding to "KILL" and so on.

The top command

There is an alternative to ps which while not as powerful has the advantage of being continually updated. This command is called "top" because it was originally designed to identify the processes at the top of the job queue.

Issuing the command "top" will replace your entire terminal with a display illustrating the current situation. If you watch it for a bit you will see that it is updated every three seconds:

```
top - 11:03:45 up 12 days, 16:22,  5 users,  load average: 1.00, 1.01, 1.00
Tasks: 107 total,   2 running, 105 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0% us,  0.1% sy, 26.0% ni, 73.7% id,  0.2% wa,  0.0% hi,  0.0% si
Mem:   2075180k total,  2024716k used,    50464k free,    76964k buffers
Swap:  2097144k total,     2692k used,  2094452k free,   750356k cached


  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND

32601 abcd3     39  19 1219m 979m 7856 R 99.9 48.4 135:45.69 java
    1 root      16   0   680  252  216 S  0.0  0.0   0:09.81 init
    2 root      RT   0     0    0    0 S  0.0  0.0   0:00.27 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.02 ksoftirqd/0
    4 root      RT   0     0    0    0 S  0.0  0.0   0:00.24 migration/1
```

The information in the top five lines is generic system information. The following block has one line per process.

We can restrict the jobs shown to a single user, typically oneself. To select a user, press the "u" key. Do not press the Return key. The blank line between the generic system information and the per-process table should prompt for the user wanted:

```
top - 12:31:10 up 12 days, 17:50,  5 users,  load average: 1.03, 1.10, 1.08
Tasks: 107 total,   2 running, 105 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.1% us,  0.4% sy, 31.3% ni, 68.2% id,  0.1% wa,  0.0% hi,  0.0% si
Mem:   2075180k total,  2024076k used,    51104k free,    71060k buffers
Swap:  2097144k total,     2692k used,  2094452k free,   734160k cached

Which user (blank for all): rjd4

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
32601 stjm2     39  19 1219m 1.0g 7856 R 99.9 49.5 228:50.24 java
    1 root      16   0   680  252  216 S  0.0  0.0   0:09.81 init
    2 root      RT   0     0    0    0 S  0.0  0.0   0:00.27 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.02 ksoftirqd/0
```

After entering the user ID, press Return to get just that user's processes:

```
top - 12:33:42 up 12 days, 17:52,  5 users,  load average: 1.00, 1.06, 1.07
Tasks: 107 total,   2 running, 105 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0% us,  0.0% sy, 27.1% ni, 72.8% id,  0.2% wa,  0.0% hi,  0.0% si
Mem:   2075180k total,  2024076k used,    51104k free,    71060k buffers
Swap:  2097144k total,     2692k used,  2094452k free,   734160k cached


  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND

10442 rjd4      15   0  4436 2044 1412 S  0.0  0.1   0:00.18 bash
 2610 rjd4      16   0  2056 1004  752 R  0.0  0.0   0:00.23 top
```

We can also change the number of seconds between updates by pressing the "s" key, the number of seconds and then Return. On a heavily used, multi-user system is is regarded as polite to increase the interval to at least 10 seconds.

To quit, press “q”. There's no need to press Return afterwards.

Press “?” to get the complete set of options:

```
Help for Interactive Commands - procps version 3.2.5
Window 1:Def: Cumulative mode Off.  System: Delay 3.0 secs; Secure mode Off.


  Z,B       Global: 'Z' change color mappings; 'B' disable/enable bold
  l,t,m     Toggle Summaries: 'l' load avg; 't' task/cpu stats; 'm' mem info
  1,I       Toggle SMP view: '1' single/separate states; 'I' Irix/Solaris mode


  f,o     . Fields/Columns: 'f' add or remove; 'o' change display order
  F or O  . Select sort field
  <,>     . Move sort field: '<' next col left; '>' next col right
  R       . Toggle normal/reverse sort
  c,i,S   . Toggle: 'c' cmd name/line; 'i' idle tasks; 'S' cumulative time
  x,y     . Toggle highlights: 'x' sort field; 'y' running tasks
  z,b     . Toggle: 'z' color/mono; 'b' bold/reverse (only if 'x' or 'y')
  u       . Show specific user only
  n or #  . Set maximum tasks displayed


  k,r       Manipulate tasks: 'k' kill; 'r' renice
  d or s    Set update interval
  W         Write configuration file
  q         Quit
          ( commands shown with '.' require a visible task display window )
Press 'h' or '?' for help with Windows,
any other key to continue
```

The watch command

We can think of the `top` command as being similar to `ps` running every few seconds. It simply has the bonus of sorting the output in order of resource use.

So how would we run a command every few seconds? Suppose instead of running top we wanted to run a specific `ps` command every 10 seconds?

Here's one way. It runs a loop inside the shell that runs for ever. Each pass of the loop clears the screen, runs `ps` and then waits (“sleeps”) for 10 seconds. Hit Ctrl+C to stop it.

```
$ while true
do
clear
ps -U rjd4 -o pid,ppid,pcpu,pmem,args
sleep 10
done
```

There is a rather more “packaged” command to do this called “`watch`”. Again, Ctrl+C breaks out.

```
$ watch ps -U rjd4 -o pid,ppid,pcpu,pmem,args
```

So what does watch offer us over the slightly longer shell script?

```
Every 2.0s: ps -U rjd4 -o pid,ppid,pcpu,pmem,args              Feb  5 13:15:02

  PID  PPID %CPU %MEM COMMAND
 8747     1  0.0  0.0 ncpd
 8779     1  0.0  0.0 ncpd
 8800  8536  0.0  0.0 -bash
22826  8800  0.0  0.0 watch ps -U rjd4 -o pid,ppid,pcpu,pmem,args
24057 22826  0.0  0.0 ps -U rjd4 -o pid,ppid,pcpu,pmem,args
```

At first glance, all it offers us is a reminder of how often the command is re-run, what the command is and what the time the command was last run across the top of the screen. The other difference is more subtle. When we kill `watch` with Ctrl+C the screen returns to how it was before the command ran. This “saved screen” model can be very useful. We can also change how often it runs with the `--interval` option

```
$ watch --interval 20 ps -U rjd4 -o pid,ppid,pcpu,pmem,args
```

The real gain that watch offers us over the shell command is that it can highlight differences. We'll start by picking a silly command whose results we can be certain will change each time: `date`.

```
$ date
Sun Feb  5 13:40:19 GMT 2006

$
```

We can run this under `watch` and get the results we would expect.

```
$ watch date
```

Now run it with an extra option on `watch`, the `--differences` option:

```
$ watch --differences date
```

Note how any output from the command that is different from the output of the previous run of the command is highlighted in the output processed by `watch`, appearing in inverse video:

```
Every 2.0s: date                                              Feb  5 13:43:31
```

```


Sun Feb  5 13:43:31 GMT 2006
```

We can go further and ask for cumulative differences to be displayed (i.e. everything that's different in the output from the first run of the command):

```
$ watch --differences=cumulative date
```

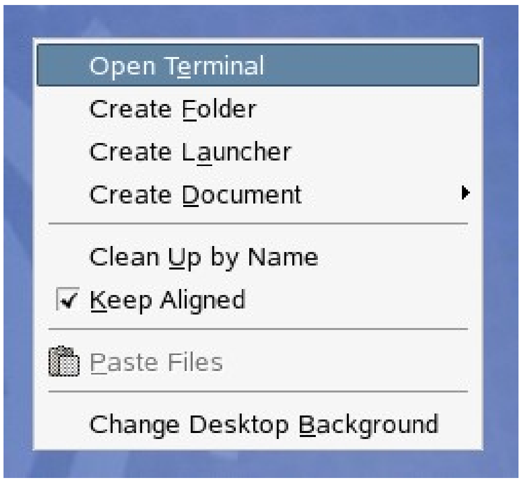
The highlighted area slowly grows over the command's output:

```
Every 2.0s: date                                     Feb  5 13:49:24


Sun Feb  5 13:49:24 GMT 2006
```

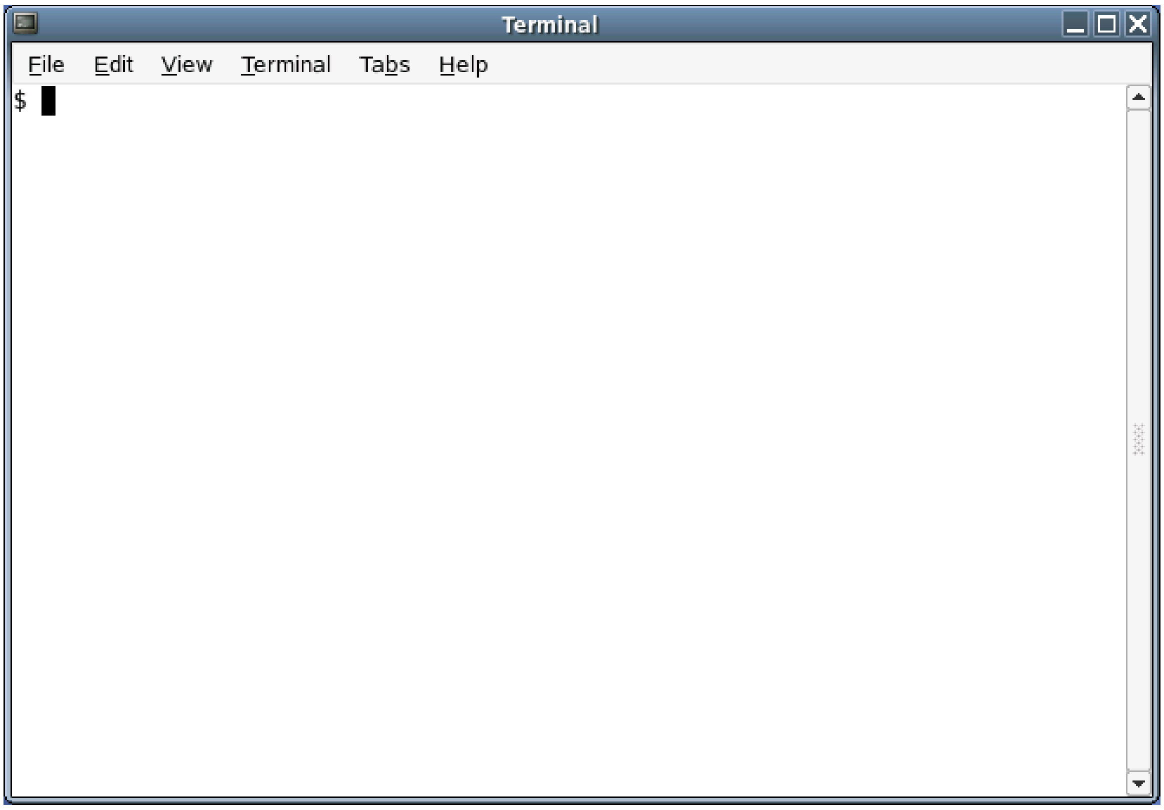
The watch command in conjunction with `ls` is also useful to spot when files have stopped growing. (e.g. log files from command runs, files being transferred in, etc.)

The terminal window



I'm using the standard “GNOME terminal window” program. There are two ways to get it on PWF Linux. The first is to right click on the background and select “Open Terminal” from the menu. (An alternative to right-clicking is to press the menu button on the keyboard while the pointer is over the background and then to select “Open Terminal” with the arrow keys and the Return key to select it.)

Alternatively, it can be selected from the Applications menu via Applications → Unix Shell → GNOME Terminal.



```
$ 
```

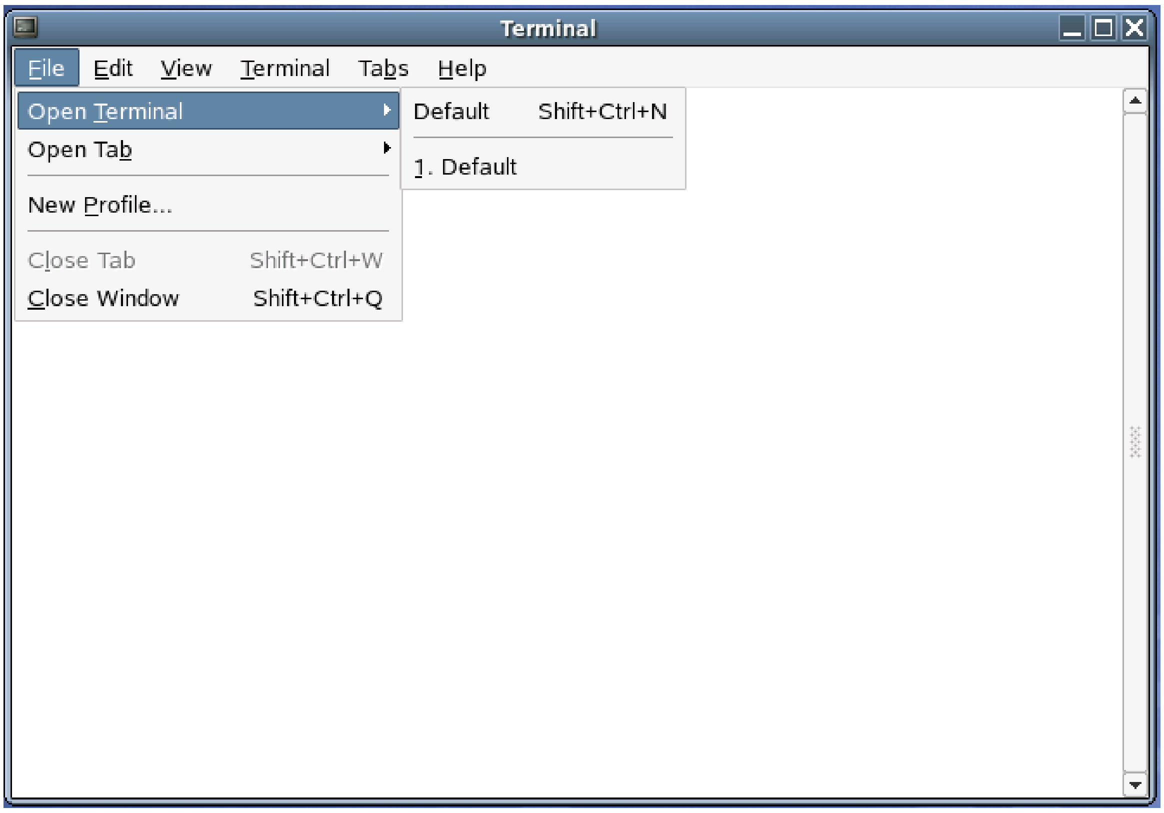
So what can the terminal emulator do?

Changing the size of text

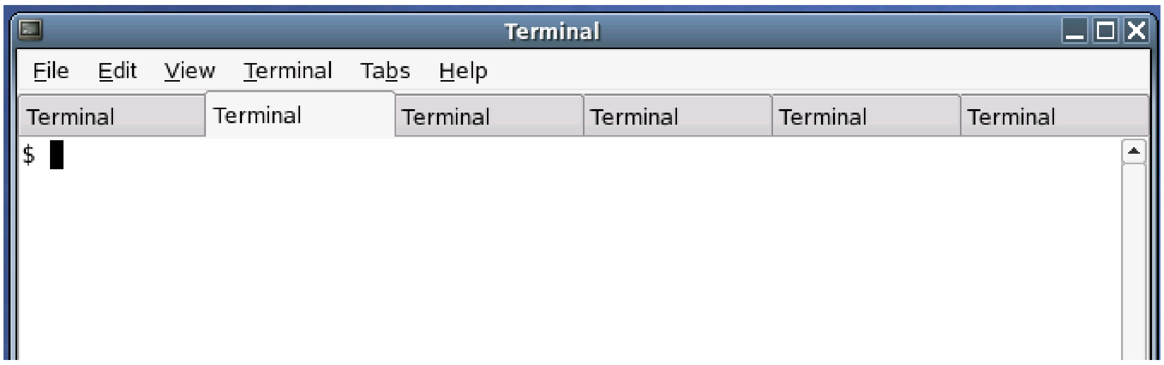
Pressing Ctrl++ makes the terminal larger and Ctrl+- makes it smaller. This has an upper limit of the height of the screen and a lower limit that is essentially unreadable. Ctrl+= will always restore it to its default size.

More terminals

You can get more terminals either by following the instructions for the first or by pressing Ctrl+Shift+n on an existing one.



More interestingly, you can get another session as a tab on the original window by pressing Ctrl+Shift+t:


Terminal
File Edit View Terminal Tabs Help
Terminal Terminal Terminal Terminal Terminal Terminal

```
$ 
```

You can switch between these by either clicking on the tab or by pressing Alt+1 for the first tab, Alt+2 for the second, etc.

Scrolling

The scroll bar on the right hand side performs as one would expect but, in addition, Shift+PgUp and Shift+PgDown will scroll up and down a windowful at a time.

Changing the prompt

The characters used by the system to prompt you to enter another command is called, unsurprisingly, the "prompt". The text used for the prompt is determined by a variable called "PS1". If we change the value of this variable the prompt changes too. Please note the importance of having some space at the end of the prompt:

```
rjd4@soup:~> export PS1=fred
fred
fred
fredpwd
/home/rjd4
fredexport PS1='fred '
fred pwd
/home/rjd4
fred
```

As well as fixed text, we can insert some special sequences starting with backslashes to generate different text according to context:

```
fred export PS1='\h:\w\$ '
soup:~$
```

Here is a list of the more useful control sequences:

	definition	e.g.
`\h`	The host name	`soup`
`\H`	The full host name	`soup.linux.pwf.cam.ac.uk`
`\t`	The time (24hr format)	`18:04:54`
`\T`	The time (12hr format)	`06:04:54`
`\u`	Current user	`rjd4`
`\w`	The full current working directory	`/home/rjd4/some/where`
`\W`	The last element of the current working directory	`where`
`\$`	A "#" if you are root, a "$" otherwise	`$`

If you put the definition in single quotes then you can include spaces in it. Anything not preceded with a backslash is taken literally.

If the export statement is put in a file "`.bashrc`" in your home directory it will be run every time you log in.

Terminal names

What is it with terminal names? If you look at the output from `ps` it quotes a terminal that the process is running with, labelling the column “`TTY`”. The entries are either “`?`” (no terminal) or “`pts/`*`N`*” or sometimes “`tty`*`N`*”. What do these mean?

The story dates back to the earlies, pre-internet days of Unix. Then a computer was a big system with large numbers of terminals directly connected to it via serial links. These were large paper-printing “teletypes”. That's what “TTY” stands for. Each of these teletypes had an explicit number and were refered to in the Unix world by the name of a corresponding device file: `/dev/tty5` say. To read what was being entered at the teletype the system would read from that file and to print text on the paper for that teletype the system would write to that file.

On a modern Linux system there are still half a dozen or so teletypes defined. These are the plain text interfaces you can get by pressing Ctrl+Alt+F*n* (for *n* between 1 and 6). Press Ctrl+alt+F7 to get back to the graphical interface. Each of those text termina lprovides an independent text login: a teletype.

The cosy world of big computing was disrupted by the arrival of the networked computer. When a connection was made to a computer remotely there was no teletype, at least not on the computer being connected to. Various tricks were attempted to get around this but ultimately the “pseudo-terminal” was arrived at. This created devices that acted as teletype devices for connections that weren't from locally attached teletypes. These are used both for network connections and sessions within windows and now form the majority of connections on a system. These pseudo-terminals come as pairs called masters and slaves and the process is attached to the slave pseudo-terminal. The device for a “pseudo-terminal slave” is a “`pts`”. Each pseudo-terminal gets a unique number and the device file for the slave is “`/dev/pts/`*`n`*” for various *n*.

If you want to know what teletype or pseudo-terminal slave you are connected to, issue the command `tty`:

```
$ tty
/dev/pts/8

$
```