UNIX, Linux, and variant history

⇒ UNIX has been a popular OS for more than two decades because of its multi-user, multi-tasking environment, stability, portability and powerful networking capabilities.

⇒ As early as 1957, Bell Labs found that they needed an operating system for their in house computer center which was running many short batch jobs, so they created BESYS to sequence the jobs and control the system resources.

⇒ By 1964, the Labs was adopting a new generation of computers and they decided to join forces with General Electric and MIT to create a new general purpose, multi-user, time-sharing operating system that they decided to call **Multics**.

⇒ However, in 1969 they decided to pull out from the project due to differences with the other members. At that point, the Labs purchased a new computer and used **GECOS**, an operating system that was not nearly as advanced as **Multics** and created the need among the research staff to create something different and more powerful.

⇒ Ken Thompson, Dennis Ritchie and others began to work on **UNIX** using an old **PDP-7** mini computer.

⇒ The name "UNIX" was intended as a pun on "Multics", and it was originally written "Unics" (**UN**iplexed **I**nformation and **C**omputing **S**ystem).

⇒ The name also intended to reflect that the new system was simpler than Multics.

UNIX Features

⇒ UNIX is a machine independent O/S (means portable – not specific to just one type of computer hardware)

⇒ UNIX is developed using high-level programming language “C” and every command is a Program.

⇒ A **multi-user** operating system is one which can support multiple people using the same system at the same time. Each user can connect to the system and run their own programs without interfering with other users. UNIX systems can easily support hundreds of users simultaneously.

⇒ **Multiprogramming** - Support more than one program, in memory, at a time. Amounts to multiple user processes on the system.

⇒ **Multitasking** - A multi-tasking operating system is one which can run multiple tasks (such as programs, applications, etc) at the same time. This is usually done by allocating a few milliseconds to each task in turn, so it appears that all the tasks are running simultaneously.

⇒ Supports **virtual memory**, programs larger than the physical RAM of the system, can be executed.

⇒ Supports a large number of tools, libraries and utilities to aid software development.

⇒ Supports hierarchical file system to hold user data organized in the form of directories and files.

⇒ Identifies a user with a userid and groupid and allows access permissions to resources to be specified using these ids.

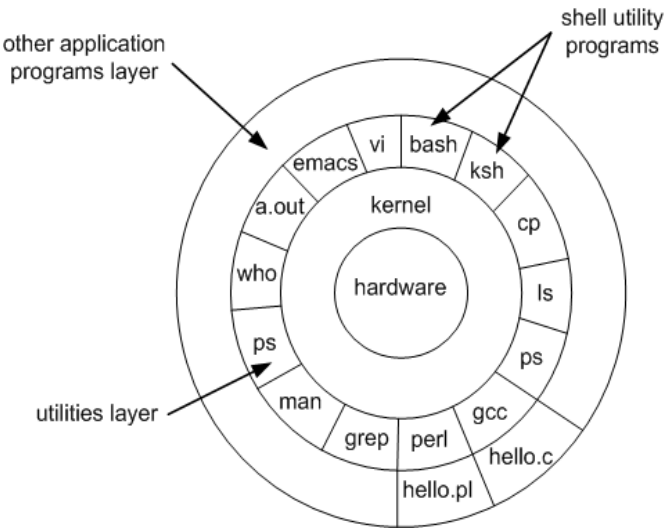
⇒ Supports a command language selectable on a per user basis. (Example: csh, sh, and ksh)

⇒ **Security:**UNIX systems provide security in several ways. Firstly, all users must identify themselves by logging in with a username and password. Once they are logged in, they may only read or write files, or run or stop programs, for which they have permission. The underlying operating system is also protected from interference by normal users. This is referred to as system security.

⇒ **Network capabilities:**UNIX systems are commonly used as servers for network services such as electronic mail, World Wide Web, file and printer sharing, and more. UNIX servers provide these services through an interface called "sockets".

Architecture/Components of UNIX

⇒ UNIX is made up of many different components (Programs) and the two main components are Kernel, and the Utilities



Kernel

⇒ The memory resident portion (raw binary form) of UNIX system (loaded into RAM by the boot loader)

The three major tasks of Kernel are: Process Management, Device Management, and File Management. (The kernel includes device driver support for a large number of PC hardware devices (graphics cards, network cards, hard disks etc.), advanced processor and memory management

⇒ features, and support for many different types of file systems (including DOS floppies and the ISO9660 standard for CDROMs.)

⇒ Three additional Services for Kernel are: Virtual Memory, Networking, and Network File Systems

⇒ Experimental Kernel Features are: Multiprocessor support, Lightweight process (thread) support

⇒ In terms of the services that it provides to application programs and system utilities, the kernel implements most BSD and SYSV system calls, as well as the system calls described in the POSIX.1 specification.

POSIX - Short for "**Portable Operating System Interface for uni-X**"

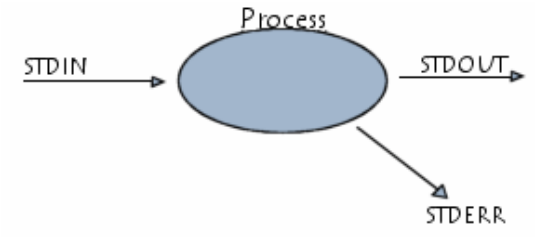
⇒ POSIX is a set of standards codified by the IEEE and issued by ANSI and ISO.

⇒ The goal of POSIX is to ease the task of cross-platform software development by establishing a set of guidelines for operating system vendors to follow. Ideally, a developer should have to write a program only once to run on all POSIX-compliant systems.

Process:

⇒ Once a command/program is run, a process is created and opens three flows:

- stdin, called the standard input, where the process will read the input data. By default stdin refers to the keyboard; STDIN is identified by the number 0;
- stdout, called standard output, where the process will write the output data. By default, stdin refers to the screen; STDOUT is identified by the number 1;
- stderr, called standard error, where the process will write error messages. By default, stderr refers to the screen. STDERR is identified by the number 2;



Hardware Control

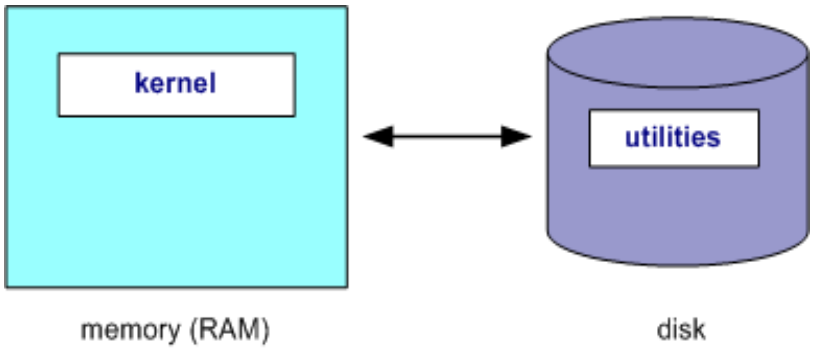
⇒ Hardware control is responsible for handling interrupts and for communicating with the machine.
⇒ Devices such as disks or terminals may interrupt the CPU while a process is executing.
⇒ The kernel may resume execution of the interrupted process after servicing the interrupt.

User mode and Kernel mode

⇒ At any given instant a computer running the Unix system is either executing a process or the kernel itself is running
⇒ The computer is in user mode when it is executing instructions in a user process and it is in kernel mode when it is executing instructions in the kernel.
⇒ Executing System call ==> User mode to Kernel mode
- perform I/O operations
- system clock interrupt

System Utilities

⇒ Virtually every system utility that you would expect to find on standard implementations of UNIX (including every system utility described in the POSIX.2 specification) has been ported to Linux.
⇒ This includes commands such as ls, cp, grep, awk, sed, bc, wc, more, and so on.
⇒ These system utilities are designed to be powerful tools that do a single task extremely well (e.g. grep finds text inside files while wc counts the number of words, lines and bytes inside a file).
⇒ Users can often solve problems by interconnecting these tools instead of writing a large monolithic application program.



⇒ Like other UNIX flavours, Linux's system utilities also include server programs called daemons which provide remote network and administration services (e.g. telnetd and sshd provide remote login facilities, lpd provides printing services, httpd serves web pages, crond runs regular system administration tasks automatically).

⇒ A daemon (probably derived from the Latin word which refers to a beneficent spirit who watches over someone, or perhaps short for "Disk And Execution MONitor") is usually spawned automatically at system startup and spends most of its time lying dormant (lurking?) waiting for some event to occur.

Note: one well known utility is the **shell.**

Application programs

⇒ Linux distributions typically come with several useful application programs as standard. Examples include the emacs editor, xv (an image viewer), gcc (a C compiler), g++ (a C++ compiler), xfig (a drawing package), latex (a powerful typesetting language) and soffice (StarOffice, which is an MS-Office style clone that can read and write Word, Excel and PowerPoint files).

⇒ Redhat Linux also comes with rpm, the Redhat Package Manager which makes it easy to install and uninstall application programs.

Logging into (and out of) UNIX Systems

Text-based (TTY) terminals:

⇒ When you connect to a UNIX computer remotely (using telnet) or when you log in locally using a text-only terminal, you will see the prompt:

login:

⇒ At this prompt, type in your usename and press the enter/return/ ↵ key.

⇒ Remember that UNIX is case sensitive (i.e. Will, WILL and will are all different logins). You should then be prompted for your password:

login: will

password:

⇒ Type your password in at the prompt and press the enter/return/ ↵ key. **Note that your password will not be displayed on the screen as you type it in.**

⇒ If you mistype your username or password you will get an appropriate message from the computer and you will be presented with the login: prompt again. Otherwise you should be presented with a shell prompt which looks something like this:

⇒ $

⇒ To log out of a text-based UNIX shell, type "**exit**" at the shell prompt (or if that doesn't work try "logout"; if that doesn't work press ctrl-d).

Graphical terminals:

⇒ If you're logging into a UNIX computer locally, or if you are using a remote login facility that supports graphics, you might instead be presented with a graphical prompt with login and password fields. Enter your user name and password in the same way as above (N.B. you may need to press the TAB key to move between fields).

⇒ Once you are logged in, you should be presented with a graphical window manager that looks similar to the Microsoft Windows interface. To bring up a window containing a shell prompt look for menus or icons which mention the words "shell", "xterm", "console" or "terminal emulator".

⇒ To log out of a graphical window manager, look for menu options similar to "Log out" or "Exit".

General format of UNIX commands

⇒ A UNIX command line consists of the name of a UNIX command (actually the "command" is the name of a built-in shell command, a system utility or an application program) followed by its "arguments" (options and the target filenames and/or expressions).

⇒ The general syntax for a UNIX command is

$ command -options targets ↵

⇒ Here **command** can be thought of as a verb, **options**as an adverb and **targets** as the direct objects of the verb. In the case that the user wishes to specify several options, these need not always be listed separately (the options can sometimes be listed altogether after a single dash).

UNIX Manual Pages:

⇒ **man:** man is the online UNIX user manual, and you can use it to get help with commands and find out about what options are supported.

⇒ It has quite a terse style which is often not that helpful, so some users prefer to the use the (non-standard) info utility if it is installed:

Example:

$ man ls

OR

$ info ls

Exercise 1:

1. Log on a Linux machine or connect to one from a Windows machine (e.g. click on the Exceed icon and then use putty to connect to the server kiwi. Enter your login (user name) and password at relevant prompts.
2. Enter these commands at the UNIX prompt, and try to interpret the output. Ask questions and don't be afraid to experiment (as a normal user you cannot do much harm):
   - echo hello world ↵
   - passwd ↵
   - date ↵
   - hostname ↵
   - arch ↵
   - uname -a ↵
   - dmesg | more ↵(you may need to press q to quit)
   - uptime ↵
   - who am i ↵
   - who ↵
   - id ↵
   - last ↵
   - finger ↵
   - w ↵
   - top ↵(you may need to press q to quit)
   - echo $SHELL ↵
   - echo {con,pre}{sent,fer}{s,ed} ↵
   - man "automatic door" ↵
   - man ls ↵(you may need to press q to quit)
   - man who ↵(you may need to press q to quit)
   - who can tell me why i got divorced ↵

- lost ↵
- clear ↵
- cal 2000 ↵
- cal 9 1752 ↵(do you notice anything unusual?)
- bc -l ↵(type quit ↵ or press Ctrl-d to quit)
- echo 5+4 | bc -l ↵
- yes please ↵(you may need to press Ctrl-c to quit)
- time sleep 5 ↵
- history ↵

Introduction to vi

⇒ **vi** (pronounced "vee-eye", short for visual, or perhaps vile) is a display-oriented text editor based on an underlying line editor called **ex**.

⇒ Although beginners usually find **vi**somewhat awkward to use, it is useful to learn because it is universally available (being supplied with all UNIX systems). It also uses standard alphanumeric keys for commands, so it can be used on almost any terminal or workstation without having to worry about unusual keyboard mappings. System administrators like users to use **vi** because it uses very few system resources.

⇒ To start vi, enter:

$ vi*filename* ↵

Where*filename* is the name of the file you want to edit. If the file doesn't exist, vi will create it for you.

⇒ The main feature that makes **vi** unique as an editor is its mode-based operation.

⇒ **vi** has two modes: **command mode** and **input mode**.

⇒ In command mode, characters you type perform actions (e.g. moving the cursor, cutting or copying text, etc.)

⇒ In input mode, characters you type are inserted or overwrite existing text.

⇒ When you begin **vi**, it is in command mode.

⇒ To put vi into input mode, press i (insert).

⇒ You can then type text which is inserted at the current cursor location; you can correct mistakes with the backspace key as you type.

⇒ To get back into command mode, press ESC (the escape key). Another way of inserting text, especially useful when you are at the end of a line is to press a (append).

Quick reference for vi editor

Inserting and typing text:

i	insert text (and enter input mode)
a	append text (to end of line)
ESC	re-enter command mode
J	join lines

File and Directory Commands

⇒ UNIX provides a number of commands for working with files.
⇒ The more common ones are described in this section.
⇒ Note that these commands usually have several options and accept wildcard characters as arguments.
⇒ For details, see the respective man pages which are hyperlinked to each command name.

pwd– print working/current directory

⇒ Displays the full absolute path to your current location in the file system.
⇒ The syntax is **pwd.** So

```
$ pwd ↵
/usr/bin
```

⇒ Implies that /usr/bin is the current working directory.

ls (list directory)

⇒ lslists the contents of a directory.
⇒ If no target directory is given, then the contents of the current working directory are displayed. So, if the current working directory is /,

```
$ ls ↵
bin    dev   home   mnt   share  usr   var
boot   etc   lib    proc  sbin   tmp   vol
```

⇒ Actually, ls doesn't show you *all* the entries in a directory - files and directories that begin with a dot (**.**) are hidden (this includes the directories '**.**' and '**..**' which are always present).

⇒ The reason for this is that files that begin with a **.**usually contain important configuration information and should not be changed under normal circumstances.

⇒ If you want to see all files, ls supports the -a option:

$ ls -a ↵

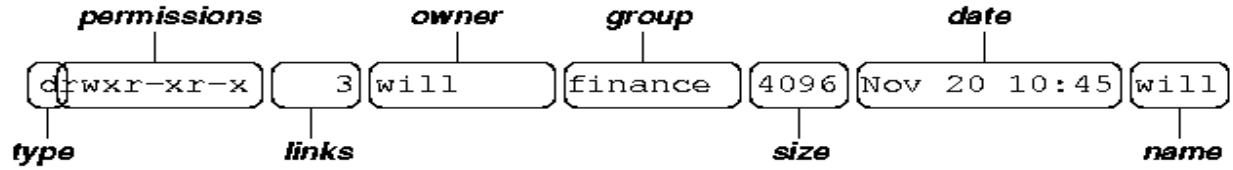
⇒ Even this listing is not that helpful - there are no hints to properties such as the size, type and ownership of files, just their names.

⇒ To see more detailed information, use the -l option (long listing), which can be combined with the -a option as follows:

$ ls -a -l ↵ (or, equivalently)

$ ls -al ↵

⇒ Each line of the output looks like this:



where

- **type** is a single character which is either 'd' (directory), '-' (ordinary file), 'l' (symbolic link), 'b' (block-oriented device) or 'c' (character-oriented device).
- **permissions**is a set of characters describing access rights. There are 9 permission characters, describing 3 access types given to 3 user categories. The three access types are read ('r'), write ('w') and execute ('x'), and the three users categories are the user who owns the file, users in the group that the file belongs to and other users (the general public). An 'r', 'w' or 'x' character means the corresponding permission is present; a '-' means it is absent.
- **links**refers to the number of filesystem links pointing to the file/directory (see the discussion on hard/soft links in the next section).
- **owner**is usually the user who created the file or directory.
- **group** denotes a collection of users who are allowed to access the file according to the group access rights specified in the permissions field.
- **size**is the length of a file, or the number of bytes used by the operating system to store the list of files in a directory.
- **date**is the date when the file or directory was last modified (written to). The -u option display the time when the file was last accessed (read).
- **name**is the name of the file or directory.

- ls -F - *mark directories with "/" and executable files with "*"*
- ls *.doc - *show all files with suffix ".doc"*

cd (change [current working] directory)

$ cd*path*

⇒ Changes your current working directory to *path* (which can be an absolute or a relative path).

⇒ One of the most common relative paths to use is '**..**' (i.e. the parent directory of the current directory).

⇒ Used without any target directory

$ cd ↵

⇒ The above command resets your current working directory to your home directory (useful if you get lost).

⇒ If you change into a directory and you subsequently want to return to your original directory, use

$ cd- ↵

Ex:

$ cd /usr/local ↵ - *change to /usr/local*
$ cd doc/training ↵ - *change to doc/training in current directory*
$ cd ~/data ↵ - *change to data directory in home directory*
$ cd ~joe ↵ - *change to user joe's home directory*

mkdir (make directory)

⇒ Creates a subdirectory called*directory*in the current working directory.

⇒ You can only create subdirectories in a directory if you have write permission on that directory.

⇒ The syntax is**mkdir *directory***

Ex:

$ mkdir /u/training/data ↵
$ mkdir data2 ↵

rmdir (remove directory)

⇒ Removes the subdirectory *directory* from the current working directory. You can only remove subdirectories if they are completely empty (i.e. of all entries besides the '.' and '..' directories).

⇒ The syntax is **rmdir directory**

Ex:

$ rmdir project1 ↵

⇒ To recursively remove nested directories, use the rm command with the -r option:

$ rm -r dirctory_name ↵

cat(catenate/type)

⇒ Displays the contents of *target-file(s)* on the screen, one after the other.
⇒ You can also use it to create files from keyboard input as follows (> is the output redirection operator). The syntax is

cat*target-file(s)*

Ex: **$ cat > hello.txt ↵**
hello world! ↵
press[ctrl-d]
$ ls hello.txt ↵
hello.txt
$ cat hello.txt ↵ - diplays entire file
hello world!
$ cat -b myprog.c ↵ - *shows line numbers*
$ cat file1 file2 > file3 ↵ - *adds file1 and file2 to make file3*

more/pg/less(catenate with pause)

⇒ Browses/displays files one screen at a time. Use h for help, spacebar to page, b for back, q to quit, /string to search for string. The syntax is

$ more*target-file(s)*

Ex: **$ more sample.f ↵**
⇒ It also incorporates a searching facility (press '/' and then type a phrase that you want to look for).
⇒ You can also use more to break up the output of commands that produce more than one screenful of output as follows (| is the pipe operator):

$ ls -l | more ↵

⇒ **less** is just like more, except that has a few extra features (such as allowing users to scroll backwards and forwards through the displayed file).
⇒ **less** not a standard utility, however and may not be present on all UNIX systems.
⇒ **pg** is similar to the more utility in function but has different commands and options. **See the man page for details.**

head- displays the first n lines of a file

head[-n] filename

where –n is number of lines, filename is the name of file which contents need to be displayed on screen.

Ex:

$ head sample.f ↵ - *display first 10 lines (default)*

$ **head -5 sample.f** ↵ - *display first 5 lines*

$head -25 sample.f ↵ - *display first 25 lines*

tail- displays the last n lines or n characters of a file

tail [-n] file name

where –n is number of lines, filename is the name of file which contents need to be displayed on screen.

Ex:

$ less sample.f ↵- *display last 10 lines (default)*
$ less -5 sample.f ↵- *display last 5 lines*
$ less -5c sample.f ↵- *display last 5 characters*
$ less -25 sample.f ↵- *display last 25 lines*

cp (copy)

⇒ cp is used to make copies of files or entire directories. To copy files, use:

cp*source-file(s) destination*

⇒ Where*source-file(s)* and *destination* specify the source and destination of the copy respectively.
⇒ The behaviour of cp depends on whether the destinationis a file or a directory.
⇒ If the destination is a file, only one source file is allowed and cp makes a new file called *destination* that has the same contents as the source file.
⇒ If the destination is a directory, many source files can be specified, each of which will be copied into the destination directory.
⇒ To copy entire directories (including their contents), use a *recursive* copy.

cp -rd *source-directories destination-directory*

Ex: $ cp sample.f sample2.f ↵ - *copies sample.f to sample2.f*

$ cp -R dir1 dir2 ↵ *- copies contents of directory dir1 to dir2*
$ cp -i file.1 file.new ↵ *- prompts if file.new will be overwritten*
$ cp *.txt chapt1 ↵ *- copies all files with .txt suffix to directory chapt1*
$ cp /usr/doc/README ~ ↵ *- copies file to your home directory*
$ cp ~betty/index . ↵ *- copies the file "index" from user betty's home directory to current directory*

mv (move/rename)

⇒ **mv** is used to rename files/directories and/or move them from one directory into another.
⇒ Exactly one source and one destination must be specified. The syntax is

mv*source destination*

⇒ If destination is an existing directory, the new name for source (whether it be a file or a directory) will be destination/source.
⇒ If source and destination are both files, source is renamed destination.
⇒ N.B.: if destination is an existing file it will be destroyed and overwritten by source (you can use the -i option if you would like to be asked for confirmation before a file is overwritten in this way).

Ex:

$ mv sample.f sample2.f ↵ *- moves sample.f to sample2.f*
$ mv dir1 newdir/dir2 ↵ *- moves contents of directory dir1 to newdir/dir2*
$ mv -i file.1 file.new ↵ *- prompts if file.new will be overwritten*
$ mv *.txt chapt1 ↵ *- moves all files with .txt suffix to directory chapt1*

rm (remove/delete)

⇒ Removes the specified files. Unlike other operating systems, it is almost impossible to recover a deleted file unless you have a backup (there is no recycle bin!) so use this command with care.
⇒ If you would like to be asked before files are deleted, use the -i option: The syntax is

rm*target-file(s)*

⇒ rm can also be used to delete directories (along with all of their contents, including any subdirectories they contain). To do this, use the -r option.
⇒ To avoid rm from asking any questions or giving errors (e.g. if the file doesn't exist) you used the -f (force) option.

⇒ Extreme care needs to be taken when using this option - consider what would happen if a system administrator was trying to delete user will's home directory and accidentally typed:

$ rm -rf / home/will ↵

(instead of rm -rf /home/will).

Ex: **$ rm sample.f** ↵ - deletes sample.f

$ rm chap?.txt ↵ - deletes all files with chap as the first four characters of their name and with .txt as the last four characters of their name

$ rm -i * ↵ - deletes all files in current directory but asks first for each file

$ rm -r /olddir ↵ - recursively removes all files in the directory olddirincluding the directory itself

file(identifies the "type" file)

⇒ Tell you if the object you are looking at is a file or if it is a directory.

⇒ The command syntax is:

file *filename*

Ex: file * ↵- *reports all files in current directory and their types. The output might appear as shown below:*

about.html:	ascii text
bin:	directory
staff.directory:	English text
bggen:	executable or object module not stripped
bmbinc:	commands text
machines.sp1:	[nt]roff, tbl, or eqn input text
man2html:	executable or object module not stripped
man2html.c:	ascii text

Finding Files

⇒ There are at least three ways to find files when you don't know their exact location: whereis, locate, find.

whereis:

⇒ Will search for the binary, source, and manual page files for a command.

⇒ The manpages tell you where whereis looks.

⇒ The syntax is

whereis [**-bmsu**] [**-BMS***directory*... **-f**] *filename* ...

-b Search only for binaries.

-m Search only for manual sections.

-s Search only for sources.

-u Search for unusual entries. A file is said to be unusual if it does not have one entry of each

requested type. Thus `**whereis -m -u ***' asks for those files in the current directory which

have no documentation.

-B Change or otherwise limit the places where **whereis** searches for binaries.

-M Change or otherwise limit the places where **whereis** searches for manual sections.

-S Change or otherwise limit the places where **whereis** searches for sources.

-f Terminate the last directory list and signals the start of file names, and *must* be used when

any of the **-B**, **-M**, or **-S** options are used.

Ex.:

⇒ Find all files in **/usr/bin** which are not documented in **/usr/man/man1** with source in **/usr/src**:

$ whereis -u -M /usr/man/man1 -S /usr/src -f * ↵

$ whereis locate ↵

$ whereis -b locate ↵

locate:

⇒ locate uses a database created by an updatedb to efficiently locate files.

⇒ Works great, assuming your database is updated often enough to be reasonable up to date.

⇒ Most boxes using locate have the updatedb occurring in cron (The cron daemon is a long-running process that executes commands at specific dates and times).

⇒ This means locate will not find files that have been created very recently.
The syntax is

locate [-d path | --database=path] [-e | --existing] [-i | --ignore-case] [--version] [--help] pattern...

Ex:

$ locate perl ↵

-- Locate file perl in the DB and Print the path.

$ locate -i perl ↵

-- Same search as above, case insensitive.

$ locate -q perl ↵

-- Run in Quiet Mode.

$ locate -n 2 perl ↵

-- Limit the no. of results shown to 2 first.

$ locate -U locater -o located ↵

-- Create index DB starting at locater and store the index file in locateDB.In the above example the system would locate perl on the local machine.

find- finds files.

⇒ find is perhaps one of the most powerful commands there is.

⇒ For just locating a file/program of a particular name, it'll definitely be slower than locate or whereis because it will search each and every path recursively from it's start point.

⇒ The syntax of this command is:

find*pathname -name filename -print*

⇒ The pathname defines the directory to start from.

⇒ Each subdirectory of this directory will be searched.

⇒ The -print option must be used to display results.

⇒ You can define the filename using wildcards.

⇒ If these are used, the filename must be placed in 'quotes'.

Ex:

$find . -name mtg_jan92 -print ↵ - looks for the file mtg_jan92 in current directory

$ **find ~/ -name README -print ↵** - looks for files called README throughout your *home* directory

$ find . -name '*.fm' -print ↵ - looks for all files with .fm suffix in current directory

$ find /usr/local -name gnu -type d -print ↵

- looks for a directory called gnu within the /usr/local directory

Diff - compares text files.

⇒ The **diff** command compares contents of two text files.
⇒ It can compare single files or the contents of directories.
⇒ The syntax is

diff [-c| -C *Lines* | -D[*String*] | -e | -f | -n | -u | -U*Lines*] [-b] [-i] [-t] [-w] *File1 File2*

diff [-h] [-b] *File1 File2*

To Sort the Contents of Directories and Compare Files That Are Different

diff [-c | -C *Lines* | -e | -f | -n | -u | -U*Lines*] [-b] [-i] [-l] [-r] [-s] [-S *File*] [-t] [-w]

Directory1 Directory2

diff [-h] [-b] *Directory1 Directory2*

⇒ If the *Directory1* and *Directory2* parameters are specified, the **diff** command compares the text files that have the same name in both directories.
⇒ Binary files that differ, common subdirectories, and files that appear in only one directory are listed.
⇒ When the **diff** command is run on regular files, and when comparing text files that differ during directory comparison, the **diff** command tells what lines must be changed in the files to make them agree.
⇒ If neither the *File1* nor *File2* parameter is a directory, then either may be given as - (minus sign), in which case the standard input is used.
⇒ If the *File1* parameter is a directory, then a file in that directory whose file name is the same as the *File2* parameter is used.

Ex.:

⇒ To compare two files, enter:

$ diff chap1.back chap1

This displays the differences between the files chap1.bak and chap1.

⇒ To compare two files while ignoring differences in the amount of white space, enter:

$ diff -w prog.c.bak prog.c

⇒ If two lines differ only in the number of spaces and tabs between words, the **diff -w** command considers them to be the same.

cmp

⇒ Compares the two files. For exmaple I have two different files fileone and filetwo.

⇒ The syntax is

$ cmp fileone filetwo ↵

⇒ if you run cmp command on similar files nothing is returned.

⇒ -s command can be used to return exit codes. i.e. return 0 if files are identical, 1 if files are different, 2 if files are inaccessible.

⇒ This following command prints a message 'no changes' if files are same

$ cmp -s fileone file1 && echo 'no changes' ↵

dircmp

⇒ Compares two directories.

⇒ Verify the output of the following command: (dirone, dirtwo are the subdirectories of your home directory)

$ dircmp dirone dirtwo ↵

Sort:

⇒ Used to sort and merge the lines in multiple files alphabetically, numerically, or otherwise and print the result to the standard output.

⇒ The sort utility is best put to use in shell scripts where sorting is necessary.

⇒ A sort command takes the following form:

sort [options] file1 file2 ...

⇒ Sorting of files is done by using a so-called sort key.

⇒ The sort key can be specified as an option, by default, the sort key is a single line from a file.

⇒ This said, the default behavior of sort is to sort its input, line by line, in alphabetical order.

⇒ Here is a small example to get us started. If we had a file poem with the following contents:

Great fleas have little fleas

upon their backs to bite'em,

And little fleas have lesser fleas,

and so ad infinitum.

And great fleas themselves, in turn,

have greater fleas to go on;

While these again have greater still,

and greater still, and so on.

Then the command

$ sort poem ↵

Would produce the output:

and greater still, and so on.

and so ad infinitum.

have greater fleas to go on;

upon their backs to bite'em,

And great fleas themselves, in turn,

And little fleas have lesser fleas,

Great fleas have little fleas

While these again have greater still,

⇒ It sorted the file poem line by line in pseudo-alphabetical order.
⇒ This order is spaces, then lowercase characters, then uppercase characters.
⇒ The sort command, as with all UNIX commands, can be invoked with many different options.
⇒ These options allow us to specify the sort key if we do not want to sort on the whole line, or to specify the way in which sort should order the lines (in numeric order of the sort keys, in dictionary order, backwards, etc).
⇒ Here are some of the most common options:

Reverse:

⇒ The -r option sorts the input in reverse order. Using this option on the poem file, we would get the expected output:

$sort -r poem ↵

Numerical Order:

⇒ The -n option tells sort command to sort the input in numeric order. That is, the input is sorted from the smallest numeric key value to the largest numeric key value. Here is a small example:

$ls -s | sort –n ↵

$ ls -s | sort –nr ↵

Case-insensitive sorting:

⇒ The -f option causes upper and lower case letters to be folded. That is, the sort will now be case insensitive.

Sorting on part of a line:

⇒ Normally, sort will sort its input on an entire line.

⇒ If we do not want to sort on the entire line, we can tell sort to sort on a specific field (by default fields are assumed to be separated by 1 or more space characters).

⇒ This is done with the +i option, where i is an integer. Hence, +0 is the beginning of the line, +1 is the second field, and so on.

⇒ If we wanted to sort the output according to the number of bytes used to store each file, we might use the command:

$ls -l | sort +4n ↵

Saving the output to a file:

⇒ The last option that is of interest to us is the -o option. Using -o we can specify a file to save the output of the sort command to. For example:

$ sort -o output_file filename ↵

⇒ This command saves its output into the file named output_file.

uniq

⇒ Removes duplicate adjacent lines from a file.

⇒ This facility is most useful when combined with sort.

⇒ The syntax is

uniq filename

Ex: $ sort input.txt | uniq > output.txt ↵

Making Hard and Soft (Symbolic) Links

⇒ Direct (hard/physical) and indirect (soft or symbolic) links from one file or directory to another can be created using the **ln** command.

⇒ The syntax is

ln*filename linkname*

⇒ creates another directory entry for *filename* called *linkname* (i.e. ***linkname* is a hard link**).

⇒ Both directory entries appear identical (and both now have a link count of 2).

⇒ If either *filename* or *linkname* is modified, the change will be reflected in the other file (since they are in fact just two different directory entries pointing to the same file).

ln -s *filename linkname*

⇒ creates a shortcut called *linkname* (i.e**. *linkname* is a soft link**). The shortcut appears as an entry with a special type ('l'):

$ ln -s hello.txt bye.txt ↵

$ ls -l bye.txt ↵

lrwxrwxrwx 1 will finance 13 bye.txt -> hello.txt

⇒ The link count of the source file remains unaffected.

⇒ Notice that the permission bits on a symbolic link are not used (always appearing as rwxrwxrwx).

⇒ Instead the permissions on the link are determined by the permissions on the target (hello.txt in this case).

⇒ Note that you can create a symbolic link to a file that doesn't exist,but not a hard link.

⇒ Another difference between the two is that you can create symbolic links across different physical disk devices or partitions, but hard links are restricted to the same disk partition.

⇒ Finally, most current UNIX implementations do not allow hard links to point to directories.

Command substitution means nothing more but to run a shell command and store its output to a variable or display back using echo command.

For example, display date and time:

echo "Today is $(date)"

OR

echo "Computer name is $(hostname)"

Syntax

You can use the grave accent (`) to perform a command substitution. The syntax is:

`command-name`

OR

$(command-name)

Command substitution in an echo command

echo "Text $(command-name)"

OR

echo -e "List of logged on users and what they are doing:\n $(w)"

Sample outputs:

```
List of logged on users and what they are doing:
  09:49:06 up  4:09,  3 users,  load average: 0.34, 0.33, 0.28
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
vivek    tty7     :0               05:40     ?     9:06m  0.09s /usr/bin/gnome-
vivek    pts/0    :0.0              07:02    0.00s  2:07m  0.13s bash
vivek    pts/2    :0.0              09:03   20:46m  0.04s  0.00s /bin/bash ./ssl
```

Command substitution and shell variables

You can store command output to a shell variable using the following syntax:

var=$(command-name)

Store current date and time to a variable called NOW:

```
NOW=$(date)
echo "$NOW"
```

Store system's host name to a variable called SERVERNAME:

```
SERVERNAME=$(hostname)
echo "Running command @ $SERVERNAME...."
```

Store current working directory name to a variable called CWD:

```
CWD=$(pwd)
cd /path/some/where/else
echo "Current dir $(pwd) and now going back to old dir .."
cd $CWD
```