LANCASTER UNIVERSITY

Computing Department

Contemporary Operating Systems
Prac 4
Unix/Linux Support Tools

Aims:

- To develop familiarity with some of the concepts of Unix internals and the tools used to administrate a Unix system, including examining processes and services.

Task:

- Attempt to use most of the tools listed to examine the system. Particularly man, ps, top, free, vmstat, strace, nice, renice, kill, iostat, stress, netstat, ldd, file, xwininfo, sysctl.
- The tools are all command line tools and should be run from inside a terminal emulator (e.g. xterm) on the vmware operating system installed on the lab machines.

Familiarity with other common command line tools is useful but not required. Understanding the shell pipe character: | and subshell (backtick) operators: ` ` or $( ) as well as at least the basics of such tools as: cat, grep, sed, wc, sort, uniq and awk is usually helpful.

NB - take your time, this is not a race… Where examples are given, you don't have to follow them – try something different if you like!

0. About this document and useful sources of information

Unix, Linux, BSD ...

A first thing to note is that this document is written about the abstract concept of Unix. Although a long time ago there was a single operating system that could be called "Unix" (The UNIX time-sharing system, owned by AT&T) the fracturing of the Unix systems in the 80's lead to the current situation where there are literally dozens of systems that can all call themselves Unix.

All these systems are related in a number of ways, mostly they fulfil the standards document known as POSIX (The Portable Operating System Interface) and include most of the "traditional" Unix tools, and code written for one will thus often work on the others.

It should also be noted the plural of "Unix" is often considered to be 'Unices', you may also see the phrase "Unixisms" to refer to a non-intuitive feature of Unix-alike systems.

This document will at times use the word Unix, normally when referring to the abstract notion of a Unix-like-OS, however the examples are drawn from GNU/Linux (a concrete implementation of the Unix kernel called Linux, usually distributed with the GNU userland tools. For simplicity sake this will be just written as "Linux") with some comments about the BSD family of Free Unices (FreeBSD, NetBSD, OpenBSD, Dragon Fly BSD) , however at times it will also reference Solaris, the Unix made by Sun Microsystems.

For details of the relationships between these operating systems see the Unix family tree: http://www.levenez.com/unix/history.html

Syntax
First a note about the syntax of this document, this document contains commands designed to be typed at your prompt will be prefixed by the traditional prompt character. This will be '`$`' for sh and bash shells, '`%`' for tcsh or csh shells.

Commands which must be run as root (or using sudo) will be prefixed with the `'#'` character (as this is the traditional root prompt).

You should not type the prompt character itself, no matter if its `$`, `%` or #

Comments will be interspersed between the lines or at the end of the lines but will be prefixed with `';;'` Do not type these comments as `;` is a special character in a number of shells. Shell commands and their output will be listed in `courier` format.

Documentation
Documentation for Unix systems is usually contained in what are called "man pages", these are the system manual. The command for dealing with this documentation is usually the man command, which is used for actually viewing the pages.

Other commands that are useful include:

apropos (man -k)	Searching the manual for information
whatis (man -f)	A summary of a command
info	The info system is similar to man, some programs are documented with info, however they should always have at least a brief man page giving a summary and giving information on where to find the full documentation.

Man pages follow a specific format, and should have the following sections (and perhaps some others): NAME, SYNOPSIS, DESCRIPTION, OPTIONS, FILES, SEE ALSO, BUGS, and AUTHOR.

The traditional method of listing Unix commands is to list the command followed by the section of the manual it is documented in in brackets. This is for clarity when the same name may exist in multiple sections of the manual For example:
sh(1)

This notation may be used below, however for clarity some program names will be listed in single quotes, e.g. 'bash', 'sh' etc. Or when there is no danger of confusion with other programs or terms they will just be written. Program names are traditionally written in the same case as they are named, which for most Unix commands is always lower case, even if they are at the start of a sentence.

Remember to consult the man pages for each command given, they should have a SEE ALSO section which may list a number of additionally useful related commands.

Also its worth using the apropos command, which searches the man-pages on the system for a keyword. For example:

```
$ apropos editor
ed (1)               - text editor
editfilenames (1)    - filename editor for crip
```

```
editor (1)           - Nano's ANOther editor, an enhanced free Pico
clone
eview (1)            - Vi IMproved, a programmers text editor
evim (1)             - Vi IMproved, a programmers text editor
```

There are many more entries than this, but this will do for an example. Also note if your system lacks apropos you can usually use: '`man -k` <keyword>' instead.

Most programs also include the ability to list their usage, this is usually from one of the following three flags (some programs output different amounts of information per-flag, with `-h` being usage, `-help` being short flags and `--help` being the long flags):

```
$ vim -h             ;; All three of these are equivalent.
$ vim -help          ;; This may not be the case for all
$ vim --help         ;; commands.
```

The usual method of obtaining a man page is to type:
```
$ man <name>
```

If you want to see the man page on that topic from a specific section use:
```
$ man <section> <name>
```

For example:
```
;; time(1) retrieves information on the time command
$ man 1 time

;; time(2) retrieves information on time.h
$ man 2 time
```

See man(1) and intro(1) for more.

Key Combinations:
Many man-pages will use similar notation for key combinations, since most of the software is controlled from the keyboard. The following examples illustrate this:

`^x`	Hold down Control and press 'x' at the same time.
`^H`	Hold down control and press 'h', this often backspaces.
`^x ^y`	Hold control and press 'x', release both then hold control and press 'y'
`ESC-c`	Press Escape then press 'c' as two separate pushes.
`C-f`	Hold control and hit 'f' then release both.
`C-f s`	Hold control and hit 'f', then release both and press 's'

Your Pager: less
Generally man-pages are viewed in a pager (a type of program for viewing long files on a terminal) called less. See less(1) for the full details, but some useful keys are listed below.

`q`	Quit less and return to the shell
`SPACE`	Go to the next screen of text
`b`	Go to the previous screen of text
`RETURN`	Advance a single line through the file
`g`	Go to the top of the file
`G`	Go to the end of the file
`/pattern`	Search forwards for the word "pattern" in the file
`?pattern`	Search backwards for the word "pattern" in the file
`n`	Jump to the next occurrence of the current search
`N`	Like 'n' but goes in the opposite direction (backwards for / searching and forwards for ? searching)

The searching features are particularly handy for navigating man-pages quickly
as they enable you to look for particular bits of information.

Further Information
A lot of what is here may be helped by a general familiarity with Unix and its shells.

http://www.lancs.ac.uk/~tipper/writing/luui/html/index.html contains some general high-level advice, and some localised advice about the Unix systems at Lancaster.

See TLDP and Google and more generalised information.

This document will assume you are familiar with generalised computer knowledge and will only document specific Unixisms that differ from the norm.

One piece of information worth repeating about shells here that everyone should know is that ^c (Control + c) will attempt to stop the currently running process in your shell and will give you your prompt back.

For more introductory and/or in-depth documentation see /usr/share/doc/ which should contain the README and HOWTO files. man pages however are a good method of finding the correct information quickly.

The Linux Documentation Project (TLDP) is also available to provide more in depth information on a large number of topics: http://www.tldp.org/

It contains a number of introductory Unix texts including:
http://www.tldp.org/HOWTO/From-PowerUp-To-Bash-Prompt-HOWTO.html
http://en.tldp.org/HOWTO/Unix-and-Internet-Fundamentals-HOWTO/
http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html

Longer documents include (Some are really quite indepth and long):
http://www.tldp.org/LDP/intro-linux/html/index.html
http://www.tldp.org/LDP/sag/html/index.html
http://www.tldp.org/LDP/tlk/tlk.html

http://www.tldp.org/LDP/lki/index.html
http://www.tldp.org/LDP/nag2/index.html

Quite a lot of man pages are also available online if you don't have a specific one on your system, see:
http://linuxcommand.org/

If you know the section and the name you can construct a URL to get you to the page, for example the sysctl page from section 8 is available at:
http://linuxcommand.org/man_pages/sysctl8.html

It should be noted that tldp.org also stock a large file containing most Linux man-pages in one bundle in a number of formats.

For higher-level information on the "Unix Philosophy" the design that grew out of the development of Unix see such documents as:
http://hebb.cis.uoguelph.ca/~dave/27320/new/unixphil.html
http://en.wikipedia.org/wiki/Unix_philosophy
http://www.catb.org/~esr/writings/taoup/html/philosophychapter.html
http://www.catb.org/~esr/writings/unix-koans/

1. Process Monitoring

Tools for monitoring
Unix has a number of commands which are both reasonably standard and also unstandardised. While the general role of each is fairly standard (and often defined by the POSIX specifications) different Unices have different extended options for each, although the GNU utilities can be compiled on pretty much every platform (including Windows machines).

The most common tools for examining the running process set include (note all the descriptions are taken from the man-pages):

`ps`	report a snapshot of the current processes.
`top`	display Linux tasks
`vmstat`	Report virtual memory statistics
`free`	Display amount of free and used memory in the system

Some Linux systems may also have commands such as:

`slabtop`	display kernel slab cache information in real time
`iostat`	Report Central Processing Unit (CPU) statistics and input/output statistics for devices and partitions.

Processes:
Commands in Unix systems are always launched by a parent process (with the exception of init(8), the parent of all processes).

The traditional model is that a parent process uses the fork(2) call to create a child process. The child process then uses exec(3) to execute the desired child process. The child process would have the PID of the parent as its PPID and would report its exit status to that PPID when it died.

For more information on init and the boot-process of a Unix system see init(8) and the much recommended HOWTO:
http://www.tldp.org/HOWTO/From-PowerUp-To-Bash-Prompt-HOWTO.html

Often the parent process will be a shell, a shell is in essence a simple process. It prints a prompt and when you hit enter it interprets what you've typed, running it as a process. When its finished it simply prints the prompt again for you to type another command.

Common shells include 'sh', 'ksh', 'csh', 'tcsh', 'bash' and more recently 'zsh'. These shells usually include internal commands designed to limit various qualities about their child processes.

The sh(1) and bash(1) shells use the ulimit command, this is an internal command to the shell and thus it doesn't have its own man-page. If you read bash(1) and search for "ulimit" then you can find the documentation for it. Alternatively you can consult the internal "help" command in a bash shell in the following manner:

```
$ help ulimit
```

Limits can also be set by the administrator of a system using such tools as quota(1) to limit disk consumption. Linux systems also generally include limits.conf(5), which is part of pam(7) (see also linux-pam(7)). Other Unices are likely to have their own methods of doing this, login(1) is likely to be the place to look for information.

Commands also have a priority for the system. This can be set at the start of the processes life by using nice(1), on many Unices there is also the renice(8) command which can be used to "alter priority of running processes"

To run a command simply type its name at the prompt, for example:

```
$ ps
```

Its worth noting that commands in Unix operate much like functions in languages such as C and thus return a numeric value. Traditionally there are two states for this value. If it is 0 then the program succeeded. If it is anything else the program failed.

Many processes include very specific numerical codes for giving feedback on their failure, however as a rule if its not 0 then it wasn't a success.

For the sh, bash, csh and tcsh shells there is a special variable that contains the return value of the last command, which is $? This is illustrated by the following examples:

```
$ ls                       ;; will print a list of files
$ echo $?                  ;; should print 0

$ ls fjkdsfjklsjdklfjksd   ;; should fail
$ echo $?                  ;; should print 1
```

An example of this is the following shell script:

```
#!/bin/bash
# check-return.sh checks the return code of ls

ls $1 > /dev/null 2> /dev/null

if [[ $? == 0 ]]; then
  echo "File exists"
  exit 0
else
  echo "File doesn't exist"
  exit 1
fi
```

This runs ls with an argument of the first argument passed to the script, it redirects the output to /dev/null, thus discarding it, and redirects the error output (2>) there as well. This means ls runs totally silently.

it then checks the return code of the last program run against being zero. If its zero it prints a “File exists” message, otherwise it notifies the user that it doesn't.

```
$ ./check-return.sh /etc/passwd
File exists

$ ./check-return.sh foobarbaz
File doesn't exist
```

Signals
Unix includes a method of communicating with its processes known as 'signals' are a part of the POSIX specification, see signal(7), signal(2) for more.

On most Unices the method of sending a signal to a command is via kill(1), which is rather erroneously named as its a method of sending any signal you wish to a process.

The kill commands default signal to send is SIGTERM (signal 15). Here the processes catches the signal and reacts to it, the traditional reaction to a SIGTERM is to quickly clean up any temporary files, write some state to disk perhaps and then exit (often with a return of 1).

To find the PID (Process IDentifier) of a process you should use the ps command. By default it will only print processes running in the current shell, however you can make it print the PIDs of all your processes by issuing it the -U flag followed by the username your interested in (your own username will be contained in the shell variable $USER and thus you can use -U $USER):

```
;; Lists all your processes
$ ps -U $USER
```

So log into a machine twice, and in one shell start a process (such as an editor) and in the other window find the PID of it, ps(1) includes a number of more efficient methods of searching than listing all your processes then looking and issue a kill. For example:

```
;; First login
$ tty  ;; Prints out this shells TTY
$ vim  ;; Starts a long running process
```

```
;; Second login
$ ps -C vim   ;; First column is PID, second TTY
$ kill <pid>  ;; Kill the PID you saw in the first command
```

For more information on shells and the idea of a TTY see:
http://dict.die.net/tty/
`man getty`
`login(1)`

More on signals: zombies, orphans and kill -9
Sometimes you will get processes that are locked up to the point of being unable to respond to a SIGTERM, or they are written to catch the SIGTERM but the code has a bug that means the process won't stop.

In this situations you need to use the SIGKILL signal (normally signal 9), this indicates to the Operating System that you would like the process to be killed by it, and not give it a chance to do anything.

```
;; Request the operating system kill the process.
$ kill -9 <pid>
```

This may lead to the following consequences: Processes may not clean up temporary files, they may not clean up file-locks (see flock(2)), child processes may become orphans.

The processes of a Unix system act as follows. They start, run, and when they finish or are issued a SIGTERM/SIGKILL they report a 1 byte response of their exit status to their Parent Process (PPID). If they get locked up trying to process a SIGTERM their status may be changed to 'z' (zombie status) and they may need to be issued a SIGKILL. However if their PPID is killed (by a user, a segfault or anything else) then they will become orphans.

Orphaned processes are inherited by init (PID 1) and have their PPID changed to 1. When they exit (SIGTERM, SIGKILL, normal exit from inside (see exit(3))) then they will report their exit status to init, which will discard it and let them die. So if that status was important in some way it will be lost if their PPID is killed (of course since the PPID was killed you have no way of collecting and processing it, thus think carefully before kill -9ing).

System Calls
If you want to monitor a process and work out what system calls with what arguments are being made by it then you will need to use a wrapper process around it.

On the Linux's and BSD's this is normally strace(1), however its worth noting that on Solaris machines strace is used to print debugging information for the STREAMS events from drivers and modules. So for Solaris (and similar Unices) you will need to use truss(1).

This can be performed simply, for example:

```
;; Prints all the system calls made by ls.
$ strace ls
```

Since strace (on Linux at least) by default prints to STDERR (The Standard Error output stream, as opposed to STDOUT, the Standard Output stream) then you cannot initially pipe it to another command. However strace(1) lists that you can do the following to pipe the output of strace into a pager such as less:

```
$ strace -o "|less" ls
```

However due to the nature of the terminal it is advised that you only use this piping method into such programs as grep as it is likely to not work cleanly with full screen pagers such as less.

It is easier to redirect the output of strace to a file:

```
$ strace -o output-file ls
$ less output-file
```

A snapshot of running processes: ps

Its worth noting that both top(1) and ps(1) allow for a number of customisations to the view. These are almost certainly implementation dependent beyond a few simple ones and thus its best to consult the man-pages for your particular OS.

However, the procps implementation (which is commonly used on many Linux systems, for more information see the procps homepage: http://procps.sourceforge.net) includes a number of useful flags, the following is taken from ps(1):

Selection

`-A`	select all processes
`-C`	select by command name
`-G`	select by RGID
`-U`	select by RUID
`-u`	select by EUID
`-t`	select by TTY

Output

`-F`	extra full format
`--forest`	ASCII art process tree (non-full command names)
`f`	ASCII art process tree (full command names with status column)
`s`	display signal format
`u`	display user-oriented format
`v`	display virtual memory format

Threading

`-L`	show threads, possibly with LWP and NLWP columns
`-T`	show threads, possibly with SPID column
`-m`	show threads after processes

Please note that to ensure backwards compatibility with a number of versions of ps the procps packages have some very complicated argument parsing, and include long options (options with double dashes proceeding them, such as --forest) which don't have a short option version (a flag with a single dash, such as -A) and they also include arguments that include **no dashes** such as 's', 'u' and 'v'

A series of examples demonstrating this are below:

```
;; select all processes
$ ps -A

;; select all processes whose process name is ls
$ ps -C ls

;; select all processes where real User ID (i.e. the one they
;; started with) is foo
$ ps -U foo

;; Select all procs where the effective UID (i.e. the UID
;; they are running as now as processes can change UID or
;; GID while running, see setuid(2), seteuid(2)) is “foo”
$ ps -u foo

;; Print out an ASCII art tree of the processes and parents
$ ps -A --forest

;; Print the "user-oriented view", all processes from that
;; UID with columns for % of CPU and Memory, Virtual and
;; Resident memory, TTY, Status, Start time and CPU time
;; consumed of each PID.
$ ps u

;; Print information about how the users processes and their
;; signals, including Pending, Blocked, Ignored and Caught.
;; See ps(1) and the source code for details.
$ ps s
```

An updating view of running processes: top

top(1) is useful in that it gives a running view of the most "expensive" (in CPU and memory terms) processes on a system that can be set to update at user defined intervals. Like ps it has a lot of options, and a lot of them are different in order to be compatible with a number of previous incarnations of top.

Generally however the most common options you will need are:

```
;; Run top in "batch" mode, so it just runs once and prints
;; its output This lists all processes.
$ top -b -n 1

;; Monitor only the processes of the user “foo”
$ top -u foo
```

Its worth noting that the information displayed by top can be customised as the default view cannot contain everything all at once. When top is running you can (in the procps version) customise what its view is by pressing the 'f' key and following the instructions.

To quit top when you are on the main display (which is where you first start) press either 'q' to quit properly or `^c` to issue a SIGTERM.

The concept of system load: uptime
At the top of the display of top(1) the “load average” is listed from three periods, 15 minutes, 5 minutes and 1 minute.

The load rating on a Unix system is generally the number of processes queuing for access to the CPU. Less than 1 is preferred, more than 5 is heavy, however systems can keep running to quite extraordinary load levels (many server systems have been known to run with a load of 2-4 or so for much of their life, although performance on them may be hampered by the efficiency of the scheduler and constant swapping).

To obtain a quick overview just of the load of a system the uptime(1) command can be run.

Threads and older Linux installations
Both top and ps however have some interesting behaviour under Linux 2.2 and 2.4 (this is fixed in 2.6) whereby since the POSIX threads library (pthreads) most commonly used is the GNU one they cannot display thread group properly, and threads are shown as their own PID (usually with TTY ?).

2. Process Priorities

Processes priority on Unix machines is usually controlled by two programs. nice(1) and renice(8).

nice is used to "run a program with modified scheduling priority". On most systems it is usual for the users to be able to run their processes with lessened priority then normal, but only root to be able to start a process with a higher priority.

To see the priority of running processes use something like this:

```
;; The NI column lists the nice value, the PRI column lists
;; its priority for the scheduler.
$ ps -e -o pid,ni,pri,command
```

If you run nice in a shell then it will print out the priority of that shell (which is inherited by all child processes). If you want to start a process with a modified priority then use the following syntax:

```
;; Amount is the priority, 19 is the lowest priority,
;; -20 is the highest.
$ nice -n <amount> <command>

;; If you try to start a process with a higher priority you
;; will probably be denied.
$ nice -n -5 ls
nice: cannot set priority: Permission denied

;; However if you want to run a process with a higher priority
;; then with root permissions do the following:
# nice -n -10 ls
```

```
;; With sudo to assume temporary root privs, see sudo(8)
$ sudo -u root nice -n -10 ls
```

renice(8) is a command that is used to change the priority of a process once its running, its syntax is largely the same:

```
;; Set <PID>'s nice value to <value>
# renice <value> <PID>

;; Bump the priority of your current shell up by 5 ($$ is
;; the current processes PID in sh and bash)
# renice -5 $$
```

3. Monitoring System Resource Usage: I/O and Memory

The memory management of Unix machines can vary quite widely, there have been several versions for Linux alone, however there are several ways to examine the amount being used.

Both ps and top include a number of columns for information on the memory subsystem. For example tops default view contains the VIRT, RES, SHR columns. These are Virtual Image, Resident Size (in core memory) and Shared Memory size (see top(1) and the source for more information).

A good tool for getting generalised information about the memory of the system is free(1), this lists the total main memory the system has as well as details of any swap partitions on the filesystem, its output is fairly self explanatory.

vmstat(8) contains much more specific information, including information on the memory (virtual and main), swap, IO and CPU of the system.

If you are interested in more in-depth information on the kernel then its worth looking (on 2.6 Linux machines) at slabtop(1) which is used to: "display kernel slab cache information in real time", slabs are caches of commonly referred to data structures in the kernel. This includes information about the virtual memory system, the file systems, networks systems, etc. See the man pages for more information.

There is also a package called sysstat (http://freshmeat.net/projects/sysstat/) that includes such useful tools as iostat(1) which is used to give information on the usage of your filesystems in terms of blocks written and read from them (along with per-second averages).

4. Artificially Creating System Load

There are several programs available to load a Unix machine (see http://www.testingfaqs.org/t-load.html) however one available for Linux is stress(1) (see http://weather.ou.edu/~apw/projects/stress/)

This can be used to load the CPU, IO or VM subsystems of a Unix machine. Its operation is fairly simple, however if you are planning on monitoring its impact with tools such as iostat or top then you may wish to start these tools before stress as otherwise they may take some time starting depending on your system.

5. Unix Services: Daemons

Most Unix machines offer a number of services either to just the users of that machine or to other machines over a network. These services are normally managed by programs called “daemons”. This is a rough definition of a daemon (see http://dict.die.net/daemon/ for more):

“A process that runs in the background of a machine, not relying on a terminal for input and output. It is usually idle (or in some cases not running) until some condition awakens it to perform its service. This can be a network connection being made, being sent a signal by another process (such as cron) or data being pushed into its input by a users process.”

See init(8), inetd(8), xinetd(8), cron(8) and fcron(8) for more information on the methods of starting a daemon, see also the SysV init scripts in /etc/init.d/

Since there are several methods of starting such a process then there are several methods of finding out what services are available on each machine.

Firstly you could read the init-scripts themselves, however this can be a complex endeavour and is not advised if you lack experience in doing such. If you are interested in doing this then see the scripts in /etc/init.d/ and examine how they interact with the /etc/rc*.d directories, also read init(8).

Secondly you could assume that they will all largely have the PPID of 1 and be children of init. This will not catch them all, however it will show a lot of the housekeeping processes that keep a system running (these are not however all daemons) as well as processes whose parent has died and have been inherited (see the discussion above of zombies).

A reasonable method of doing this is as follows:

```
;; filter the output of ps with grep(1) (note that this will
;; often be described or documented as "grep for...") for all
;; processes with 1 in the third column (the PPID).
$ ps --no-heading -Af | egrep "^.* .* 1 .*$"


;; This second example uses awk for cleaner filtering, it should
;; print the PIDs of all processes whose PPID is 1 and the
;; command used to run them (seperated by a gap).
$ ps h -e -o ppid,pid,cmd | awk '$1 == 1 {print $2 "\t" $3}'
```

Failing that you may be interested in those processes that are listening on network ports. This information can be obtained from the netstat program, the following commands work on Linux machines.

```
;; List all listening (-l) processes, and list the network
;; port/socket they listen on numerically (-n)
;;
;; Remove the -n flag for names to be written in full.
;;
;; See /etc/services for the full list of services and
;; port-numbers known to your system.
```

```
$ netstat -ln

;; List only those in the address family (-A) of inet, which is
;; IPv4 Networking
$ netstat -ln -A inet

;; List only those in the inet6 (IPv6) family
$ netstat -ln -A inet6
```

With these methods combined as well as your knowledge of ps and top you should be able to find all the services on a common Unix system.

6. Distinguishing Files: file

Files on Unix systems are not typed by their file extensions, they are instead characterised by a series of things such as the "magic number" (magic(5)) at the start of them, their format, path, file-extension etc.

They can all however be treated as byte streams (which is usually the safest way) even the special files that represent devices can be read with textual tools such as cat(1) and grep(1) (although this isn't advised) and to some extent directories (the original idea of a directory on a Unix system was essentially a file whose contents were the contents of the directory). It is perhaps easier to say that everything on a Unix system is an inode (an entry in the file system).

The usual tool for identifying the specific type of file on a Unix system is file(1). This relies on a number of methods, including (in this order): filesystem tests, magic number tests, and language tests. Generally the magic number entries should include one (or more) of three keywords: text, executable and data (see the man pages for more discussion of this).

Here are some examples from a Linux system, the entries are the command followed by the output from file.

```
$ file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.2.0, dynamically linked (uses shared libs), stripped

$ file /lib/ld-linux.so.2
/lib/ld-linux.so.2: symbolic link to `ld-2.3.2.so'

$ file /lib/ld-2.3.5.so
/lib/ld-2.3.5.so: ELF 32-bit LSB shared object, Intel 80386, version
1 (SYSV), stripped

$ file /etc/services
/etc/services: ASCII English text

$ file /home
/home: directory

$ file /usr/bin/cpan
/usr/bin/cpan: perl script text executable
```

7. Shared Libraries: ldd

The tool most often used to examine binaries (see file(1)) and "print shared library dependencies" is ldd(1). When run it takes a single argument of a path to a binary, it will then print out the libraries they are linked against followed by the path to that library on your system.

Normally binaries simple name the library (e.g libz.so.1) and then rely on the linker to find the library on your system (see ldconfig(8)), the output for this style of linking looks like the following:

```
        libz.so.1 => /usr/lib/libz.so.1 (0x40955000)
```

However some binaries are linked to static paths during their compilation, this is normally only done for very specific reasons (such as being the dynamic link library itself, which must be in a fixed place to be found). Heres an example:

```
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

(As far as I know ld-linux.so.2 is the dynamic linker for Linux systems (see ld.so(8)), on Solaris boxes this is known as libdl.so.1, on BSDs machines I'm not sure).

To get the listing of libraries that a binary relies on use the following syntax:

```
$ ldd <path>

;; Example, get the libraries required for ls
$ ldd /bin/ls

;; The libraries required for vi, this requires a bash(1) shell
;; for sh(1) shells use:   `which vim`
$ ldd $(which vim)
```

If ldd cannot find the correct library it will print a "not found" message after the arrow.

8. Graphical User Interfaces: The X Window System

If you have a GUI on a Unix box it is almost certainly a version of X (the X Window System), older releases were maintained by the X Consortium, then the system forked into Xfree86 and X/Open, and finally recombined into the newer X.org Foundation. If your version is X11R6.7 or later then it is a release of the X.org Foundation.

X is too large and complex for an in-depth understanding to be given here, a brief view is as follows:

X in itself is a graphical server, it provides an abstraction to the systems hardware. The applications that are run and interacted with by the users are the clients for X, and they communicate over traditional BSD sockets (see socket(2)), this automatically allows for network transparency.

However X doesn't manage where to draw the windows on the screen, clients don't particularly mind, they just need to know how big their area is. So there is a third program which is used called a Window Manager. This sits between the X-client and the X-server and does all the fiddly work. Applications draw just the internals of their windows, X draws graphics on the screen only where told, the Window Manager (a specialised X-client) draws the widgets around each application window and manages communicating when a user "drags" a window about to move or resize it.

See the following for more information:
http://www.x.org/
http://www.tldp.org/HOWTO/XWindow-Overview-HOWTO/index.html
http://xorg.freedesktop.org/releases/X11R7.0/doc/html/index.html
http://pdo.debian.net/stable/x11/xspecs

If you want to find out information about an X-client you need to use a simple command line program, xwininfo(1). A typical usage is as follows:

```
$ xwininfo

xwininfo: Please select the window about which you
          would like information by clicking the
          mouse in that window.

xwininfo: Window id: 0xe00010 "user@cent1:~"

  Absolute upper-left X:  516
  Absolute upper-left Y:  26
  Relative upper-left X:  4
  Relative upper-left Y:  26
  Width: 505
  Height: 342
  Depth: 24
  Visual Class: TrueColor
  Border width: 0
  Class: InputOutput
  Colormap: 0x20 (installed)
  Bit Gravity State: NorthWestGravity
  Window Gravity State: NorthWestGravity
  Backing Store State: NotUseful
  Save Under State: no
  Map State: IsViewable
  Override Redirect State: no
  Corners:  +516+26  -3+26  -3-400  +516-400
  -geometry 81x26-0+0
```

After the command is launched you should see your cursor change into a cross-hair, at which point clicking on a window will cause xwininfo to spit out all the information about it, as you can see.

9. The Kernel Exposed: /proc and /dev

The Linux Kernel is a monolithic kernel (almost) which has a system for loading and unloading binary modules while it is running. See the following for more information:
http://www.kernel.org/
http://www.linux.org/
http://www.tldp.org/LDP/lki/index.html
http://www.tldp.org/LDP/tlk/tlk.html

Two interesting parts of the Unix kernel are the two most common pseudo filesystems, proc(5) and dev (The /dev filesystem has no one man-page but most of section 4 of the manual is dedicated to it).

These are both connected with the Unix philosophy of "everything is a file" or at least accesible and processable using the normal file and text based tools.

The hardware devices: /dev

dev was created so that the hardware components of the system and also external devices could be accessed or viewed as part of the Unix file system. It contains a number of files, each of these represents a device, or a potential device which may be connected to the system and are linked to the drivers for those devices by the kernel of the OS.

Many of the devices are numbered so that multiple can be connected, for example there may be a /dev/lp0 and a /dev/lp1, however since these devices are part of the file system then a default can be chosen simply by symlinking (see symlink(2) and ln(1)) one of these to /dev/lp, which many programs will then write too. The idea being that most of these devices can just have streams of bytes read from them or pushed to them to accomplish interesting behaviour, with the kernel drivers translating between the OS and the hardware.

Common entries in /dev includes:

`/dev/lp0`	Line Printer 0, attached to a printer usually, see lp(4)
`/dev/ttyS1`	The second serial port on the machine
`/dev/hda`	The whole of the Hard Disk Drive (HDD) attached to the master of the first IDE channel.
`/dev/hda3`	The 3rd partition of the first HDD
`/dev/sdb6`	The 6th partition of the second SCSI or SATA device
`/dev/pts/*`	The directory of pseudo-tty slave devices. How the OS understands, tracks and communicates with logged in terminals. See pts(4)
`/dev/mem`	A device that is an image of the systems main memory, it should be handled carefully. See mem(4). There is also a /dev/kmem that is the kernels view of virtual memory.
`/dev/null`	The bit bucket, anything written to here is simply discarded by the operating system. See null(4) reading from here always returns EOF (End-Of-File).
`/dev/zero`	Reading from this device returns an infinite supply to 0s. See zero(4).
`/dev/random`	Either a software RNG from the kernel or attached to a hardware RNG when strong randomness is needed (e.g. for certain levels of encryption) see random(4).

An example of the use of device files is as follows:

```
;; See dd(1) copies the entire of hda, including its
;; file-system to a single file for examination and dissecton
# dd if=/dev/hda1 of=foo

;; View whatever ascii characters are encoded in the filesystem
$ strings foo | less

;; Blanks an entire harddisk by writing 0s to it.  Be very careful.
;; You'll need mkfs(8) to recover from this.
# dd if=/dev/zero of=/dev/hda

;; A random number generator using shell programs only
;;
;; Use od(1) to read 3 bytes from /dev/random, interpretting
;; them as long signed decimals.
;;
;; Pipe the output from that into head, selecting the first
;; line only
;;
;; Print the second column.
$ od -N 3 -l /dev/random | head -n 1 | awk '{print $2}'
```

The Kernel and Processes: /proc
/proc is designed to shift the kernel-space into the "everything is a file" mentality of Unix, it contains an interface to the currently running system kernel and allows you to read information about various settings and processes with the standard text tools.

For example if you use the following you can find out information about your system:

```
$ cat /proc/cpuinfo /proc/version /proc/mounts
```

Much of this information can be obtained from other tools, such as uname(1), mount(1) etc.

However the /proc file system is actually governed by the normal permission system of the filesystem and a number of files in it are writeable for root. This means you can actually change a number of settings of a currently running kernel by hand. For example to turn on the setting that governs if your machine can forward IP packets (useful for a number of networking tasks) you can do the following:

```
$ echo "1" > /proc/sys/net/ipv4/ip_forward
```

However there is an easier way, a program designed to do this kind of thing for you (and thus reduce the danger of typos), this is sysctl(8).

```
;; Any user can get sysctl to print out its current settings,
;; you only need root permissions to set them.
$ sysctl -a
```

Documentation for all the possible keys to set for a running kernel is sparse, since most of the people who write scripts using sysctl are kernel hackers adjusting various parameters for testing purposes. However documentation is available in a number of places.

See the following for more information:

sysctl(8)
The comments in: /usr/include/linux/sysctl.h

http://www.idevelopment.info/data/Unix/General_UNIX/GENERAL_TheProcFilesystem.sht
ml
http://www.linuxexposed.com/Articles/General/Virtual-File-System---proc.html
http://www.opennet.ru/base/dev/procdoc.txt.html