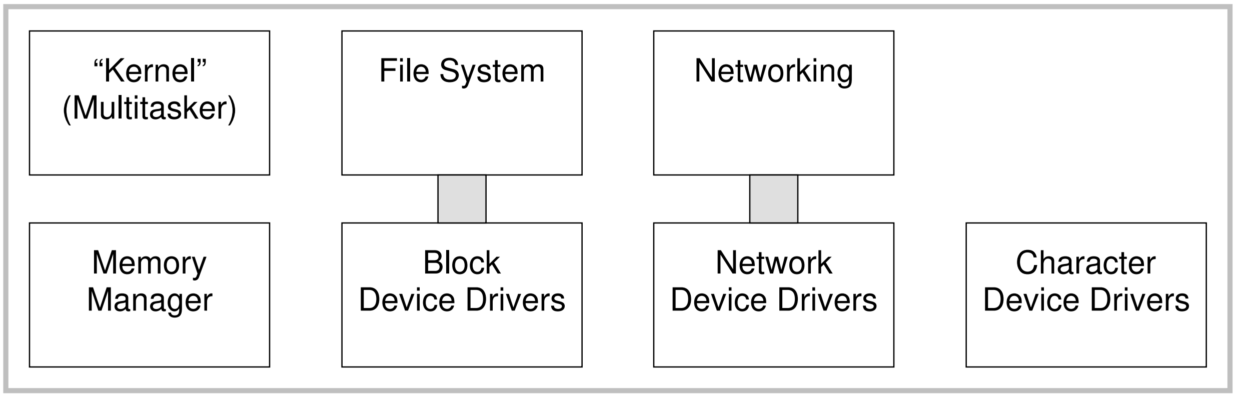
1. Overview



The **KERNEL**, as its name implies, is the core of the Linux operating system. Its main functions are:

- *Multitasking,* sharing the processor's time between multiple programs *(processes)*
- *Memory Management*, including *virtual memory* (transparently swapping blocks of memory between RAM and disk to give the appearance of additional RAM)
- Providing a *File System* for organising secondary storage
- *Networking* (TCP/IP)
- *Device Drivers* to communicate with block devices (i.e. disks), network and other hardware

What this boils down to is that the kernel is responsible for *sharing the resources* of the computer (such as the processor, RAM, hard drives etc.) *in a controlled manner*. Some of these 'controls' are common-sense limitations, such as the memory manager not letting one process read or write blocks of memory allocated to another process; others reflect policy from the system supervisor and users themselves, such as one user not being allowed to write to another user's files. Either way, these controls prevent faulty or malicious programs from causing damage, so we can add another important function to this list:

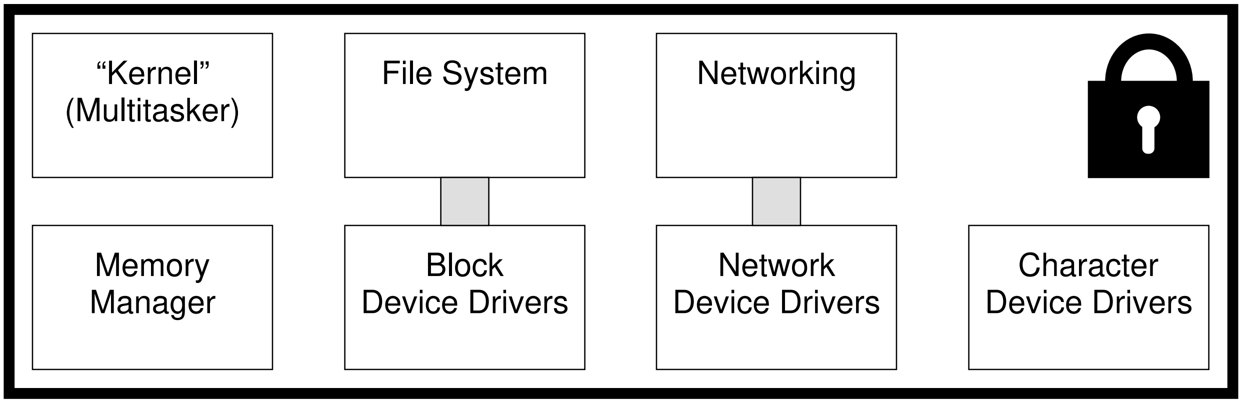
- *Security (system integrity; access restrictions)*

A comparison of these features against MS-DOS shows that Linux has some big advantages. MS-DOS only really provides a file system (hence the name – Disk Operating System) and a very primitive form of memory management. Security features are non-existent, as are multitasking and networking. Only networking can be easily added.

Important Commands

`man` *`command`*	Display information on *command* from the on-line manual. Not much use unless you know the name of the command you need!
`man -k` *`text`*	Search the manual index for commands with *text* in their description
`man -a` *`command`*	Show matching manual pages in *all* sections of the manual. Otherwise, only the first match is shown; e.g. **man -a mount** shows the manual pages in sections 2 and 8, while **man mount** shows only mount(2)

2. Security



Before we look at the components of the kernel, we need to take a brief look at the security systems which are in place.

Users are identified by means of a **user id (uid)**, and also have a **group id (gid)**. These are represented internally as numbers, but are normally mapped to names when they are displayed. This mapping is kept in the file **`/etc/passwd`**, along with other information[1].

The 'super user' on the system has a *uid* of zero, and is called **root**. When a program is run by *root*, all access controls are bypassed – it is given full access to all devices and files[2]. The upshot of this is that being logged in as *root* is **very dangerous!** A slip of the key, a mistyped command asking to delete all files instead of a certain range, can cause serious loss of data on your system. There is no 'undelete' in Linux. It is therefore important, even if you are the administrator of a system, to have a *normal, unprivileged* user account to do most of your work, and only log in as *root* when absolutely essential. On the other hand, when you are logged in as a normal user, the damage you can do is limited so feel free to experiment with all the commands. Commands which you *have* to be *root* to run are indicated thus: ✪.

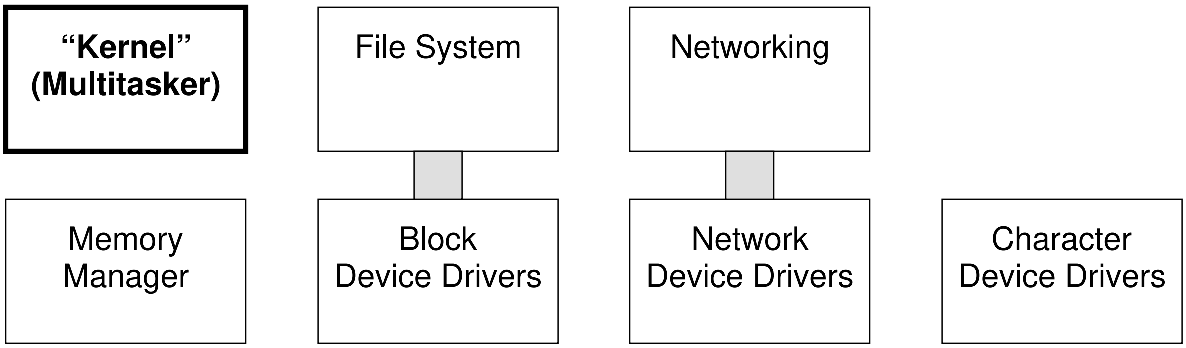
Finally, the *root* password is the ultimate security key. Unless you want every would-be hacker in the neighbourhood interfering with your system, make sure you set this password!!

Important Commands

`whoami`	Show what user name you are logged in as
`passwd`	Change your password
✪ **`useradd -m`** *`name`* ✪ **`passwd`** *`name`*	Create a new user account and set its password (on some Linux systems, type **adduser** and follow the prompts)
`man -a passwd`	Information about the *passwd* command and the */etc/passwd* file

[1] The mapping of *gids* to group names is in /etc/group
[2] Certain restrictions, such as two processes not accessing the same serial port, do still apply

3. Processes



A crucial part of the kernel is responsible for sharing the processor between multiple tasks, called *processes*. I shall call this the *multitasker*. Because of the central role it plays, confusingly it is also referred to as the 'kernel'.

The distinction between a *program* and a *process* is that a process is a particular *active instance* of a program. Consider that more than one user may be using a program at a given time; each instance is a *process*, runs independently of the others, and has its own data storage workspace (although it can share the same copy of the actual program code). At the other extreme, no-one may be using the program at all, in which case there will be no processes running it. (Actually this is something of an oversimplification – one program may divide into multiple processes if that helps it do its job. This is known as *spawning*)

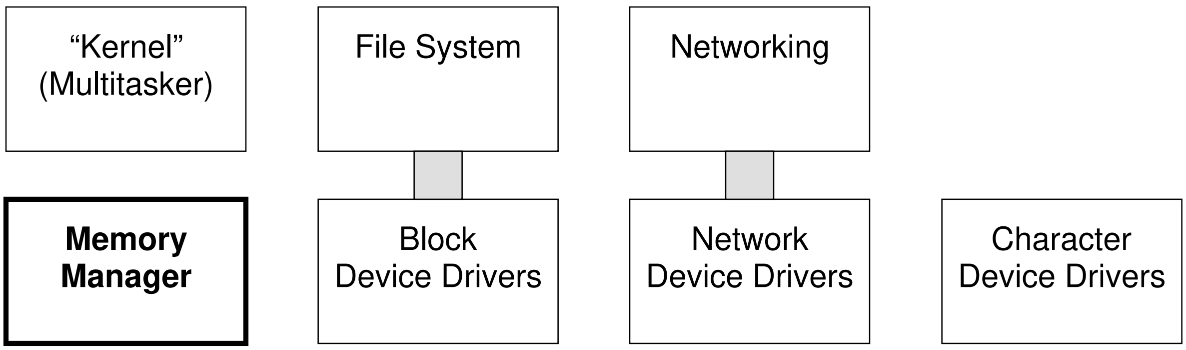
When one process is unable to continue, perhaps because it is waiting for keyboard input or a disk operation, or it has simply used up its allocated share of processor time, it is put on hold and another process is allowed to continue. This is done transparently, so each process believes it is running on the machine by itself – except somewhat slower, depending on the workload.

Normally processes are creating when needed, and vanish when they have finished their useful work. Sometimes it is necessary to 'kill' an unwanted process by hand though.

Important Commands *(remember: for more information use the **man** command, e.g. **man ps**)*

`ps -aux`	Show all processes in the process table in detailled format
`kill` *n*	Send a signal to process number *n* that it should *TERMinate*. A process may *trap* this signal to give it a chance to tidy up and terminate gracefully; otherwise, the kernel will kill it immediately.
`kill -9` *n*	Send a process a *KILL* signal, which unlike *TERM,* it can't trap. Use this if a process refuses to die after given a *TERM*. Don't use it as a matter of course, because it doesn't give the process a chance to clean up.
`kill -1` *n*	*(number 'one')* Send a *HUP (Hang-Up)* signal to the process. Many system programs use this as an instruction to re-read their config files.
`kill -l`	*(letter 'ell')* List all possible signals

4. Memory Manager



The next most important resource in the computer which needs to be shared is the RAM (Random Access Memory). Processes can make requests of the kernel whenever they need RAM, and the kernel will provide it – assuming it has not run out! The memory manager protects this memory; if a program ever tries to access memory outside of its allocated space, usually as a result of a program bug, it is immediately terminated and a *Segment Violation Error* is displayed. This protects the integrity of other processes on the system. The hardware to be able to do this was only introduced in the 80386, so Linux requires a 386 or later PC.

Since RAM is an expensive resource, and often many processes are asleep for long periods of time anyway, Linux provides a way to use areas of disk to emulate RAM. This is known as *virtual memory*. RAM is broken up into blocks called *pages*; whenever a new page is needed, and none is free, the least-recently-used page is *swapped* (copied) out to disk and thus made free for reuse. If a program ever tries to access a page which has been swapped out, the kernel transparently goes and fetches it immediately. Swapping may be done either to a dedicated partition on a hard drive, or to a file; usually the partition method is preferable. Processes whose workspace has been swapped out to disk are indicated in the output of the **ps** command by having their names in parentheses.

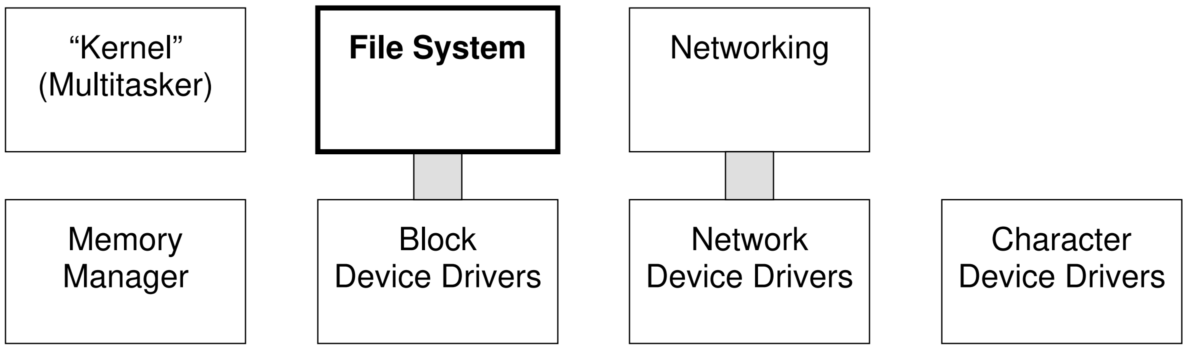
A minimal Linux system needs at least 4MB of *real* RAM to run, and I'd recommend 8MB of swap space. To run X-Windows or big 'C' compilations acceptably, 8MB of real RAM is needed and 8 to 16MB of swap. Any spare RAM is automatically used as disk cache, by the way.

Important Commands

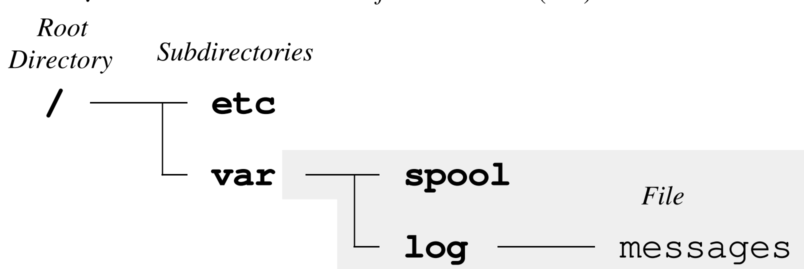
	free	Display a summary of RAM and swap space usage
✪	**swapon** *device*	Use a device or file (which must have been already prepared) for swap
✪	**swapoff** *device*	Stop using a swap device or file
☆	**mkswap** ...	Prepare a swap device or file *(see **man** page and Linux installation instructions)*

☆ indicates a command which, although it may be run by unprivileged users in certain circumstances, it is classified as a 'system program' and is stored under **/sbin** or **/usr/sbin**. Normal users need to give the full path, e.g. **/sbin/mkswap**

5. File System – Introduction



With Linux, data on block devices (such as hard drives and CD-ROMs) is organised into a single, unified *file system* or *file tree*. The file system starts at a *root directory*, identified by a forward slash '**`/`**'. Directories can contain *files* and other (sub)directories.



A file is identified by a 'path name' made up of the directories enclosing it plus its name, separated by slashes – e.g. **`/var/log/messages`** in the example above. This is analogous to the MS-DOS file system, except with forward slashes instead of backslashes.

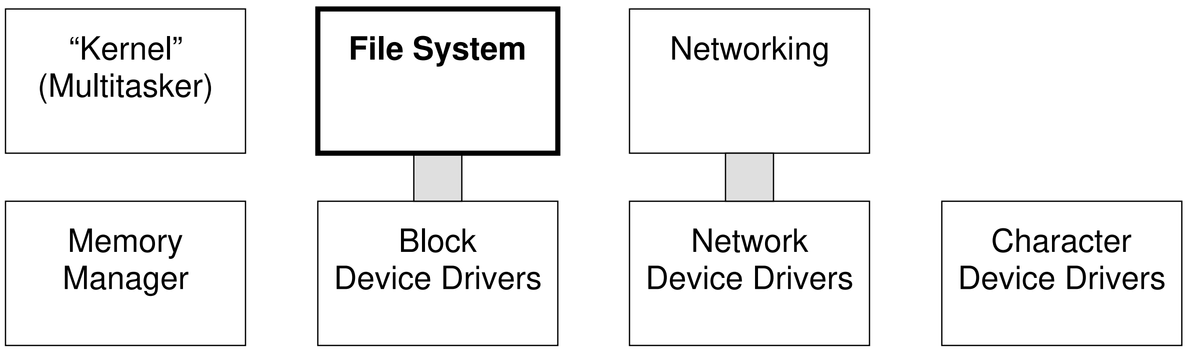
However there is a fundamental difference. MS-DOS refers to different physical devices by a driver letter (**`A:`**, **`C:`** etc) and each has its own root directory. Linux's file system has only *one* root directory, which is on the *root device* (chosen at boot time). File systems on other devices are *mounted* on the main file tree as if they were subdirectories. For example, in the diagram above, **/var** (and its subdirectories) are located on a different drive to the rest of the file system, which is what the shading is meant to indicate.

Because the Linux file tree is actually composed of a number of separate file systems, it is called a *virtual file system*.

Important Commands

`mount`	List all devices and where they are on the virtual file system
`df`	Display **d**isk **f**ree space and usage stats for each mounted device
`find / -type d`	List all directories

6. File System – Basic Operations



In order to make life easier, Linux maintains the following information for each process:

• A *working (or current) directory*. If you specify a filename *without* a leading slash, then the working directory is prepended to the front. This can save a great deal of typing! e.g. if the working directory is **`/var/spool/uucppublic`** and you type **`rm myfile`**, this is the same as typing **`rm /var/spool/uucppublic/myfile`**

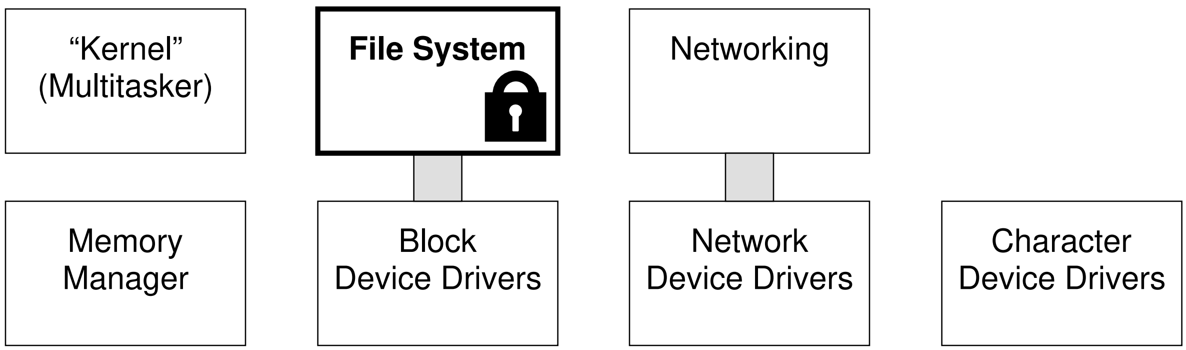
• A *home directory*. This is the initial value of the working directory. Also, you can specify your home directory in path names as **`~`**, and someone else's home directory as **`~`***`user`*. So if your home directory is **`/home/chris`**, then **`~/thesis`** is the same as **`/home/chris/thesis`** [1]

Important Commands

`ls` *`{directory}`*	List contents of the given directory, or working directory if none specified
`cd` *`{directory}`*	Change working directory (to your home directory if none specified)
`cd ..`	Move up one directory level towards the root
`pwd`	Print (i.e. display) the current working directory
`mkdir` *`directory`*	Make a directory
`rmdir` *`directory`*	Remove a directory *(must be empty)*
`cp` *`file1 file2`*	Copy *'file1'*, naming the copy *'file2'*
`cp` *`file{s} dir`*	Copy one or more files into another directory; each copy has the same name as the original
`mv` *`file1 file2`*	Move a file (or rename it, if the destination is in the same directory)
`mv` *`file{s} dir`*	Move one or more files with their existing names into another directory
`rm` *`file{s}`*	Permanently remove (delete) the given files

[1] Home directory mapping is done by the shell, not the kernel. Hence there are occasional circumstances when it doesn't work as expected. The home directory is determined from a field in **`/etc/passwd`**

7. File System – Security



Access to files is controlled by the kernel. Each file and directory has an **owner** (user id) and **group** (group id), so that accesses can be divided into three different classes – those by the owner, those in the file's group, and everyone else. In this way, certain types of access can be restricted to a specific user or group of users.

The operations which are permitted are defined by **mode bits**. For a file these are:

- **r** File can be Read
- **w** File can be Written (modified)
- **x** File can be eXecuted (i.e. is a program)

For a directory, these bits have somewhat different meanings:

- **r** Directory contents can be listed (e.g. **ls**)
- **w** Files within this directory can be created, deleted or renamed
- **x** Directory can be entered (e.g. **cd**)

There are three sets of *rwx* bits for each file – those which apply to the file's owner, those which apply to the file's group, and those which apply to everyone else. For example:

Type (e.g. **d** *= directory)*

File owner can read and write

File group can read only

`-rw-r-----`

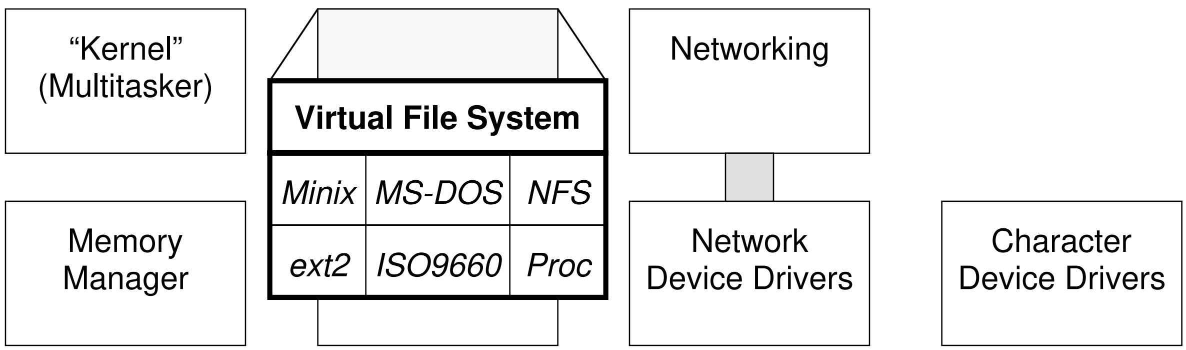
Everyone else has no access

For convenience, mode bits are often given numerically in octal (with r=4, w=2, x=1). Hence the mode bits for this example are 640.

Important Commands

`ls -l` *`{files}`*	Show permissions of a file or files
`ls -ld` *`{directory}`*	Show permissions of directory itself, rather than files it contains
✪ **`chown`** *`user file(s)`*	Change owner of given files or directories
✪ **`chgrp`** *`group file(s)`*	Change group
✪ **`chown`** *`user.group files`*	Change owner and group at same time
`chmod` *`perms files`*	Change mode bits. *Perms* can be symbolic or octal

8. File System – VFS and Mount



There are a number of different types of file system which might be on a device – that is, different ways of organising the data. Linux supports several different types, all of which form part of the *virtual file system* so the applications need not worry what type of file system a file is actually stored on. The main block file systems are:

• *Minix* – standard, used for exchanging data with other Unix systems *(e.g. on floppy)*
• *Ext2* (Second Extended Filesystem2) – the preferred Linux hard disk file system, with high performance and support for long filenames
• *MSDOS and VFAT (Windows 95 long filenames)*
• *ISO9660* (used on CD-ROMs)

There are two other filesystems which are somewhat different:
• *NFS* (Network File System) – instead of using a local block device, this file system uses kernel networking to store and retrieve files on a remote server
• *Proc* – this is not a real file system, but a means by which the kernel publicises certain information about itself to the outside world. Programs such as **ps** display this information.

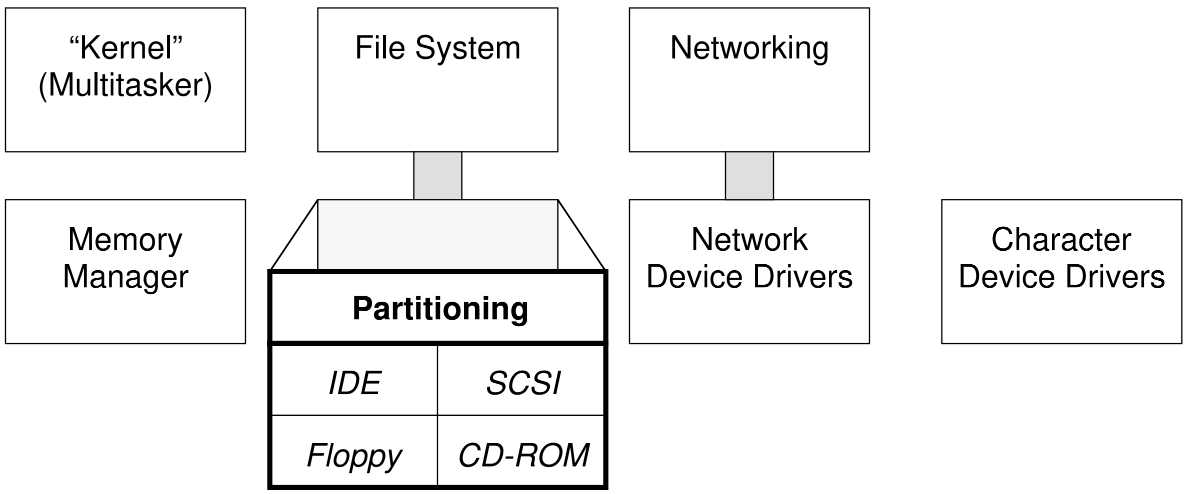
A device is attached to the virtual file system by means of the **mount** command. It is attached at a *mount point* which is basically an empty subdirectory. After mounting, the contents of the file system are visible inside that subdirectory.

The device must have a valid file system on it *before* you attempt to mount it. Hence when starting with a new device or partition, a valid file system must be created on it first.

Important Commands

✪ `mount -t msdos /dev/fd0 /a`	Mount the MS-DOS file system on the floppy in drive 0 (A:) onto empty subdirectory `/a`
✪ `umount /dev/fd0` *or* `umount /a`	Unmount this floppy disk *(you MUST do this before removing the disk, to prevent corruption)*
✪ `mount -t iso9660 /dev/scd0 /mnt/cdrom`	Mount SCSI CD-ROM onto `/mnt/cdrom`
✪ `mkfs -c -t ext2 /dev/hda2`	Make a blank ext2 file system on `/dev/hda2`. *Dangerous – will erase ALL data on this partition!!*

9. Block Devices



A *block device* is one where the storage space is divided into a number of equally-sized blocks (usually 512 or 1024 bytes) which can be read in any order. Hard drives, floppy drives and CD-ROMs fall into this category.

A hard drive is subdivided into contiguous storage areas called *partitions*, and each of these is treated as if it were a separate device; that is, each partition may contain a different type of file system, or not be used for a file system at all (e.g. it could be a swap partition)

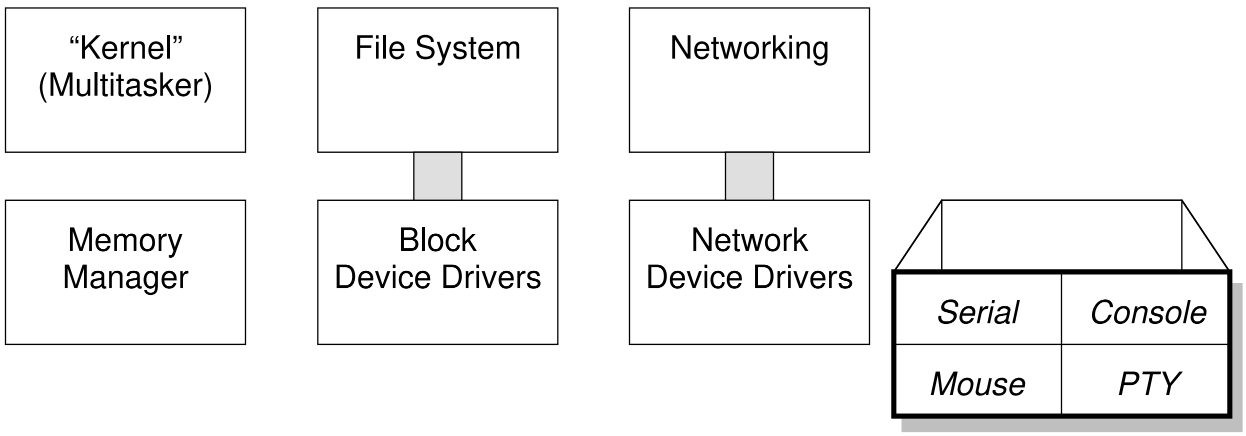
The kernel communicates with physical devices through software *device drivers*. Devices are internally identified in the kernel by a *major and minor device number*, giving the type of the device and one particular instance of it. However to the outside world, devices are referenced by names in the **`/dev`** directory, e.g. **`/dev/hda2`** for the second partition on the first IDE hard drive. This eliminates kernel-specific numbers from user programs, and gives a convenient method of access control for devices, using the file system's own access controls.

`/dev/hda`	First IDE hard drive *(entire)*	**`/dev/sda`**	First SCSI hard drive
`/dev/hda1`	First partition on hda	**`/dev/sda1`**	First partition on sda
`/dev/hda2`	Second partition on hda *etc*	**`/dev/scd0`**	First SCSI CD-ROM
`/dev/hdb`	Second IDE hard drive	**`/dev/mcd`**	Mitsumi (non-IDE) CD-ROM
`/dev/fd0`	First floppy drive	**`/dev/sbpcd`**	Soundblaster (non-IDE) CD-ROM

Important Commands

**`ls -l /dev	less`**	Show all possible block/character devices; the type is given by the first letter on each line (**b** or **c**) and major/minor device numbers are shown
✪ **`fdisk /dev/hda`**	Display or set partition information for first IDE hard drive. **Very dangerous** – you can easily wipe the whole drive…

10. Character Devices



Character devices input or output streams of data; they include all non-storage I/O devices (e.g. keyboard, screen, mouse, serial ports), as well as sequential access storage media (tape drives). They are also accessed through files in /dev, and it is possible to read or write data directly to or from them, if you have sufficient permissions on the dev file.

Normally you don't need to access character devices directly, because when you log in, the terminal you are on is automatically connected to the shell or whatever other program you are using. If you log in over the network, a "pseudo-tty" is created for the duration of the session with a name like **/dev/ttyp0**; this is like a real character device, except it is associated with a TCP/IP stream rather than a physical port on your computer.

There are also some kernel devices which you can use. **/dev/null** throws away any data that you send to it; for example, if you don't want the output of a command to appear on the screen, you can redirect it to /dev/null. **/dev/zero** gives an infinite stream of zeros when you read from it, which can be useful for initialising files.

`/dev/tty1…8`	Virtual consoles	`/dev/ttyS0…3`	Serial ports (dial-in mode)
`/dev/lp0…2`	Parallel ports	`/dev/cua0…3`	Serial ports (dial-out mode)
`/dev/psaux`	PS/2 mouse	`/dev/dsp`	Sound card PCM audio

Important Commands

`cat myfile >/dev/lp1`	Print this file (bypassing the lpd daemon)
`cu -l cua0 -s 38400`	Chat to modem on COM1 at 38400 bps
`tail -f /var/log/messages >/dev/tty8 &`	Display the last 10 lines of this file on virtual console 8 (Alt-F8) then any new lines as the file grows
`dd if=/dev/zero of=temp bs=1k count=32`	Create file 'temp' containing 32K of zeros