Operating Systems: Unix/Linux

Unix/Linux

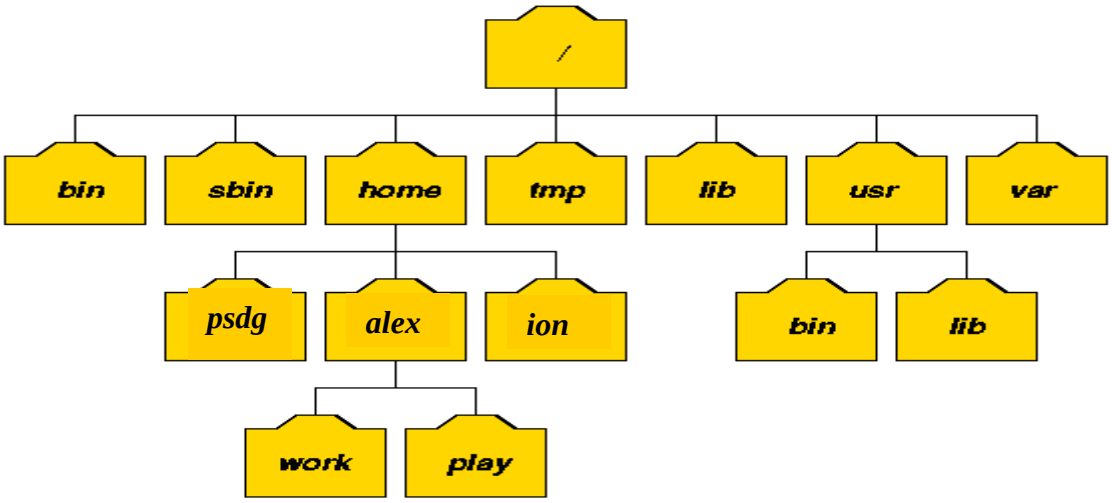
Unix Filesystem: The UNIX operating system is built around the concept of a filesystem which is used to store all of the information that constitutes the long-term state of the system. This state includes the operating system kernel itself, the executable files for the commands supported by the operating system, configuration information, temporary workfiles, user data, and various special files that are used to give controlled access to system hardware and operating system functions.

Every item stored in a UNIX filesystem belongs to one of four types:

1. **Ordinary files** : Ordinary files can contain text, data, or program information. Files cannot contain other files or directories. Unlike other operating systems, UNIX filenames are not broken into a name part and an extension part (although extensions are still frequently used as a means to classify files). Instead they can contain any keyboard character except for '/' and be up to 256 characters long (note however that characters such as *,?,# and & have special meaning in most shells and should not therefore be used in filenames). Putting spaces in filenames also makes them difficult to manipulate - rather use the underscore '_'.
2. **Directories** : Directories are containers or folders that hold files, and other directories.
3. **Devices** : To provide applications with easy access to hardware devices, UNIX allows them to be used in much the same way as ordinary files. There are two types of devices in UNIX - **block-oriented** devices which transfer data in blocks (e.g. hard disks) and **character-oriented** devices that transfer data on a byte-by-byte basis (e.g. modems and dumb terminals).
4. **Links** : A link is a pointer to another file. There are two types of links - a **hard link** to a file is indistinguishable from the file itself. A **soft link** (or symbolic link) provides an indirect pointer or shortcut to a file. A soft link is implemented as a directory file entry containing a pathname.

Typical Unix Directory Structure

The UNIX filesystem is laid out as a hierarchical tree structure which is anchored at a special top-level directory known as the root (designated by a slash '/'). Because of the tree structure, a directory can have many child directories, but only one parent directory. Figure below illustrates this layout.



To specify a location in the directory hierarchy, we must specify a path through the tree. The path to a location can be defined by an absolute path from the root /, or as a relative path from the current working directory. To specify a path, each directory along the route from the source to the destination must be included in the path, with each directory in the sequence being separated by a slash. To help with the specification of relative paths, UNIX provides

the shorthand "." for the current directory and ".." for the parent directory. For example, the absolute path to the directory "play" is /home/alex/play, while the relative path to this directory from "psdg" is ../alex/play. Note that these although these subdirectories appear as part of a seamless logical filesystem, they do not need be present on the same hard disk device; some may even be located on a remote machine and accessed across a network.
Q. Compare and contrast the relative and absolute pathnames.

Directory	Typical Contents
/	The "root" directory
/bin	Essential low-level system utilities
/usr/bin	Higher-level system utilities and application programs
/sbin	Superuser system utilities (for performing system administration tasks)
/lib	Program libraries (collections of system calls that can be included in programs by a compiler) for low-level system utilities
/usr/lib	Program libraries for higher-level user programs
/tmp	Temporary file storage space (can be used by any user)
/home or /homes	User home directories containing personal file space for each user. Each directory is named after the login of the user.
/etc	UNIX system configuration and information files
/dev	Hardware devices
/proc	A pseudo-filesystem which is used as an interface to the kernel. Includes a sub-directory for each active program (or process).

Typical UNIX directories

When one logs into UNIX, the current working directory is the user home directory. The home directory can be referred to as "~" and the home directory of other users as "~<login>". So ~alex/play is another way for user *psdg* to specify an absolute path to the directory /homes/alex/play. User alex may refer to the directory as ~/play

- pwd (print [current] working directory)

pwd displays the full absolute path to the current location in the filesystem. So

```
$ pwd ↵
/usr/bin
```

implies that /usr/bin is the current working directory.

- ls (list directory)

ls lists 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
```

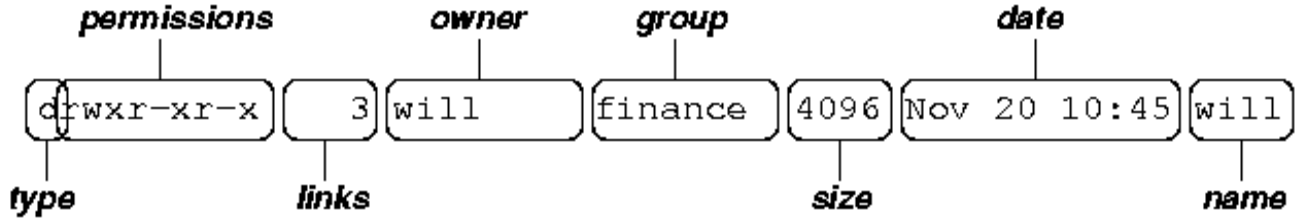
Actually, ls doesn't show *all* the entries in a directory - files and directories that begin with a dot (.) are hidden (this includes the directories '.' and '..' which are always present). The reason for this is that files that begin with a . usually contain important configuration information and 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.
- *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 supports more options. To find out what they are, type:

$ man ls ↵

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:

$ info ls ↵

- 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 ↵

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 - ↵

- mkdir (make directory)

$ mkdir *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.

- rmdir (remove directory)

  $ rmdir *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).

- 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 destination is 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*

- 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:

  $ 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).

- rm (remove/delete)

  $ rm *target-file(s)*

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:

  $ rm -i myfile ↵
  rm: remove 'myfile'?

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).

- cat (catenate/type)

  $ cat *target-file(s)*

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).

  $ cat > hello.txt ↵
  hello world! ↵
  [ctrl-d]
  $ ls hello.txt ↵
  hello.txt
  $ cat hello.txt ↵
  hello world!
  $

- more and less (catenate with pause)

$ more *target-file(s)*

displays the contents of *target-file(s)* on the screen, pausing at the end of each screenful and asking the user to press a key (useful for long files). 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.

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

$ 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 alex 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.

1. ??? matches all three-character filenames.
2. ?ell? matches any five-character filenames with 'ell' in the middle.
3. he* matches any filename beginning with 'he'.
4. [m-z]*[a-l] matches any filename that begins with a letter from 'm' to 'z' and ends in a letter from 'a' to 'l'.
5. {/usr,}{/bin,/lib}/file expands to /usr/bin/file /usr/lib/file /bin/file and /lib/file.

Permission	**File**	**Directory**
read	User can look at the contents of the file	User can list the files in the directory
write	User can modify the contents of the file	User can create new files and remove existing files in the directory
execute	User can use the filename as a UNIX command	User can change into the directory, but cannot list the files unless (s)he has read permission. User can read files if (s)he has read permission on them.

Interpretation of permissions for files and directories

Every file or directory on a UNIX system has three types of permissions, describing what operations can be performed on it by various categories of users. The permissions are read (r), write (w) and execute (x), and the three categories of users are user/owner (u), group (g) and others (o). Because files and directories are different entities, the interpretation of the permissions assigned to each differs slightly.

File and directory permissions can only be modified by their owners, or by the superuser (root), by using the chmod system utility.

- chmod (change [file or directory] mode)

$ chmod *options files*

chmod accepts options in two forms. Firstly, permissions may be specified as a sequence of 3 octal digits (octal is like decimal except that the digit range is 0 to 7 instead of 0 to 9). Each octal digit represents the access permissions for the user/owner, group and others respectively. The mappings of permissions onto their corresponding octal digits is as follows:

---	**0**
--x	**1**
-w-	**2**
-wx	**3**
r--	**4**
r-x	**5**
rw-	**6**
rwx	**7**

For example the command:

$ chmod 600 private.txt

sets the permissions on private.txt to rw------- (i.e. only the owner can read and write to the file).
Permissions may be specified symbolically, using the symbols u (user), g (group), o (other), a (all), r (read), w (write), x (execute), + (add permission), - (take away permission) and = (assign permission). For example, the command:

$ chmod ug=rw,o-rw,a-x *.txt

sets the permissions on all files ending in *.txt to rw-rw---- (i.e. the owner and users in the file's group can read and write to the file, while the general public do not have any sort of access).
chmod also supports a -R option which can be used to recursively modify file permissions, e.g.

$ chmod -R go+r play

will grant group and other read rights to the directory play and all of the files and directories within play.

- chgrp (change group)

$ chgrp *group files*

can be used to change the group that a file or directory belongs to. It also supports a -R option.

Besides cat there are several other useful utilities for investigating the contents of files:

- file *filename(s)*

file analyzes a file's contents for you and reports a high-level description of what type of file it appears to be:

```
$ file myprog.c letter.txt webpage.html ↵
myprog.c:      C program text
letter.txt:    English text
webpage.html:  HTML document text
```

file can identify a wide range of files but sometimes gets understandably confused (e.g. when trying to automatically detect the difference between C++ and Java code).

- head, tail *filename*

head and tail display the first and last few lines in a file respectively. You can specify the number of lines as an option, e.g.

```
$ tail -20 messages.txt ↵
$ head -5 messages.txt ↵
```

tail includes a useful -f option that can be used to continuously monitor the last few lines of a (possibly changing) file. This can be used to monitor log files, for example:

$ tail -f /var/log/messages ↵

continuously outputs the latest additions to the system log file.

Linux Virtual File System

What is its utility? Linux can be run as a process under Windows, etc.?

- Versatile and powerful file handling facility, designed to support a wide variety of file management systems and file structures.
- VFS presents a single, uniform file system interface to user processes.
- It defines a common file model that is capable of representing any conceivable file system's general feature and behavior.
- Assumes files are objects in a computer's mass storage that share basic properties regardless of the target file system or the underlying processor hardware.
- Files have symbolic names that allow them to be uniquely identified within a specific directory in the file system.
- A file has several properties including an owner, protection against unauthorized access or modification.

The output from programs is usually written to the screen, while their input usually comes from the keyboard (if no file arguments are given). In technical terms, we say that processes usually write to **standard output** (the screen) and take their input from **standard input** (the keyboard). There is in fact another output channel called **standard error**, where processes write their error messages; by default error messages are also sent to the screen.
To redirect standard output to a file instead of the screen, we use the > operator:

```
$ echo hello ↵
hello
$ echo hello > output ↵
$ cat output ↵
hello
```

In this case, the contents of the file output will be destroyed if the file already exists. If instead we want to append the output of the echo command to the file, we can use the >> operator:

```
$ echo bye >> output ↵
$ cat output ↵
hello
bye
```

To capture standard error, prefix the > operator with a 2 (in UNIX the file numbers 0, 1 and 2 are assigned to standard input, standard output and standard error respectively), e.g.:

```
$ cat nonexistent 2>errors ↵
$ cat errors ↵
cat: nonexistent: No such file or directory
$
```

You can redirect standard error and standard output to two different files:

```
$ find . -print 1>errors 2>files ↵
```

or to the same file:

```
$ find . -print 1>output 2>output ↵
```

or

```
$ find . -print >& output ↵
```

Standard input can also be redirected using the < operator, so that input is read from a file instead of the keyboard:

```
$ cat < output ↵
hello
bye
```

You can combine input redirection with output redirection, but be careful not to use the same filename in both places. For example:

```
$ cat < output > output ↵
```

will destroy the contents of the file output. This is because the first thing the shell does when it sees the > operator is to create an empty file ready for the output.

One last point to note is that we can pass standard output to system utilities that require filenames as "-":

Introduction
A filesystem is the "method" used to organise data on a disk. It controls the allocation of disk space to files, and associates each file with a filename, directory, permissions, and other information. Unlike most other operating systems, Linux supports a wide variety of filesystems. This is possible because, in its infancy, Linux got a "virtual file system" layer that allowed any filesystem to be "plugged in". Its own standard filesystem is ext2, which is a very well-established, stable and mature filesystem. However Linux also allows you to access Windows FAT partitions, NTFS partitions etc. Now, recent versions of Linux have added new "journaling" filesystems which offer a number of advantages over the traditional ext2. The choice of filesystem is an important one since it affects performance, recovery from errors, compatibility with other OS's, limitations on partition and file sizes, and so on. With Linux distributions such as Mandrake or SuSE, every one of the filesystems described are available on installation, so one is spoilt for choice. This article explains some of the differences between the different filesystem types, to help you make the best choice for your Linux system.

Standard partitions

ext2

The ext2 filesystem is the default filesystem for Linux. It supports partitions of up to 4 Terabytes in size (1 Terabyte is 1024 Gigabytes), while a single file can be up to 2 Gigabytes. However many kernels only support block devices up to 2TB, so the practical limit is 2TB. Filenames can be up to 255 characters long. In the case of a power outage or a similar situation in which the system is switched off without a proper shutdown, the disks can be left in an "unclean" state, where some data is only partly written or where data has been written without the directory entry being updated. In that situation, Linux performs a filesystem check. The ext2 filesystem has a tried and tested filesystem checker, e2fsck. In order to ensure that everything is well, the system also runs this program once every so many boot-ups, even when correctly shut down. The ext2 filesystem also offers the possibility of undeleting files which were deleted by mistake. This is unrelated to the "Move to Trash" method of deletion which is common with many graphical environments like KDE. In the latter case, the files are not really deleted but moved to a special folder, from which they are deleted after a set limit is reached. However with ext2, even when a file is actually deleted, it is possible to restore it unless it has been overwritten. The ext2 filesystem also allows "secure deletion", which ensures that a confidential file cannot be undeleted. Special attributes allow files to be marked as unmodifiable, or append-only.

Linux Swap

The Linux swap partition is something you generally create once and then forget about. This is an amount of disk space in which Linux temporarily writes data from RAM to free up memory for other processes. The swap partition is different from all others in that it is not used to store files in, so it won't be dealt with in any further detail here. Windows and OS/2 partitions FAT partitions (vfat) FAT16 was the standard partition type up to Windows 95, and FAT32 partitions are the standard with Windows 98. FAT16 partitions are supported by all versions of Windows, while FAT32 is supported by Windows95 second edition onwards, and Windows 2000 onwards (but is not supported in Windows NT). Linux supports both reading from, and writing to FAT16 and FAT32 partitions. Their main use in a Linux context is to share files between Linux and Windows on a dual-boot system. However note that these filesystems are unable to hold information such as file owners and permissions. FAT16 partitions are limited to a maximum of 2Gb. Although the theoretical maximum size for FAT32 is 8 TB, Windows98's scandisk only supports 128Gb, and Windows2000 does not permit the creation of FAT32 disks larger than 32Gb.

NTFS

NTFS was introduced with Windows NT v4.0, and aimed at removing the limitations in the old FAT16 filesystems while adding new features not found in OS/2's HPFS. Under Linux, this filesystem is currently supported in read-only mode (note that not all Linux distributions enable this feature). Read-write access to this disk will probably be added in future. NTFS partitions are not accessible from Windows 95 or 98.

Journaled filesystems

Introduction

Whenever a computer is switched off without a proper shutdown there is the possibility that data on the disk becomes corrupted - that is, some of the data will have been written while some has not, leaving files or even

internal filesystem data in a "half-finished" state. Whenever that happens the system goes through a routine to check the disk for errors - "fsck" in Linux and "scandisk" in Windows. This is time-consuming, especially on today's very large disks. This check is also forced once every so many boot-ups, to make sure everything is working properly. Journaling filesystems get rid of these problems. Instead of writing modified files directly onto their area on the disk, the system maintains a "journal" on the disk which describes all the changes which must be made to disk. Then, a background process takes each journal entry, makes the change and marks it as completed. If the system is halted without a shutdown, any pending changes are performed when it is restarted and the system is ready to continue running in seconds. Incomplete entries in the journal are discarded. This guarantees consistency and removes the need for a long and complex filesystem check on bootup.

ext3
Ext3 is the descendant of ext2, as its name implies. In fact, it is essentially ext2 with added support for journaling. Ext3 has a significant advantage over the other options described below: It is backwards compatible. Ext2 partitions can be converted to ext3 and vice-versa without reformatting the partition. An Ext3 partition can be mounted by an older kernel with no ext3 support - it is just seen as a normal ext2 partition. Ext3 partitions, like ext2, allow files to be undeleted.

Init

The kernel, once it is loaded, finds **init** in sbin and executes it.When **init** starts, it becomes the parent or grandparent of all of the processes that start up automatically on your Linux system. The first thing **init** does, is reading its initialization file, /etc/inittab. This instructs **init** to read an initial configuration script for the environment, which sets the path, starts swapping, checks the file systems, and so on. Basically, this step takes care of everything that your system needs to have done at system initialization: setting the clock, initializing serial ports and so forth. Then **init** continues to read the /etc/inittab file, which describes how the system should be set up in each run level and sets the default *run level*. A run level is a configuration of processes. All UNIX-like systems can be run in different process configurations, such as the single user mode, which is referred to as run level 1 or run level S (or s). In this mode, only the system administrator can connect to the system. It is used to perform maintenance tasks without risks of damaging the system or user data. Naturally, in this configuration we don't need to offer user services, so they will all be disabled. Another run level is the reboot run level, or run level 6, which shuts down all running services according to the appropriate procedures and then restarts the system. Commonly, run level 3 is configured to be text mode on a Linux machine, and run level 5 initializes the graphical login and environment. After having determined the default run level for your system, **init** starts all of the background processes necessary for the system to run by looking in the appropriate rc directory for that run level. **init** runs each of the kill scripts (their file names start with a K) with a stop parameter. It then runs all of the start scripts (their file names start with an S) in the appropriate run level directory so that all services and applications are started correctly. In fact, you can execute these same scripts manually after the system is finished booting with a command like **/etc/init.d/httpd *stop*** or **service *httpd stop*** logged in as *root*, in this case stopping the web server. None of the scripts that actually start and stop the services are located in /etc/rc<x>.d. Rather, all of the files in /etc/rc<x>.d are symbolic links that point to the actual scripts located in /etc/init.d. A symbolic link is nothing more than a file that points to another file, and is used in this case because it can be created and deleted without affecting the actual scripts that kill or start the services. The symbolic links to the various scripts are numbered in a particular order so that they start in that order. You can change the order in which the services start up or are killed by changing the name of the symbolic link that refers to the script that actually controls the service. You can use the same number multiple times if you want a particular service started or stopped right before or after another service, as in the example below, listing the content of /etc/rc5.d, where **crond** and **xfs** are both started from a linkname starting with "S90". In this case, the scripts are started in alphabetical order.
After **init** has progressed through the run levels to get to the default run level, the /etc/inittab script forks a **getty** process for each virtual console (login prompt in text mode). **getty** opens tty lines, sets their modes, prints the login prompt, gets the user's name, and then initiates a login process for that user. This allows users to authenticate themselves to the system and use it. By default, most systems offer 6 virtual consoles, but as you can see from the inittab file, this is configurable.

The idea behind operating different services at different run levels essentially revolves around the fact that different systems can be used in different ways. Some services cannot be used until the system is in a particular state, or *mode*, such as being ready for more than one user or having networking available. There are times in which you

may want to operate the system in a lower mode. Examples are fixing disk corruption problems in run level 1 so no other users can possibly be on the system, or leaving a server in run level 3 without an X session running. In these cases, running services that depend upon a higher system mode to function does not make sense because they will not work correctly anyway. By already having each service assigned to start when its particular run level is reached, you ensure an orderly start up process, and you can quickly change the mode of the machine without worrying about which services to manually start or stop. Available run levels are generally described in /etc/inittab, which is partially shown below:

```
#
# inittab   This file describes how the INIT process should set up
#           the system in a certain run-level.

# Default runlevel. The runlevels are:
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS
#       (The same as 3, if you do not have networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
```

A simple description of the UNIX system, also applicable to Linux, is this:
"On a UNIX system, everything is a file; if something is not a file, it is a process."
This statement is true because there are special files that are more than just files (named pipes and sockets, for instance), but to keep things simple, saying that everything is a file is an acceptable generalization. A Linux system, just like UNIX, makes no difference between a file and a directory, since a directory is just a file containing names of other files. Programs, services, texts, images, and so forth, are all files. Input and output devices, and generally all devices, are considered to be files, according to the system.
In order to manage all those files in an orderly fashion, man likes to think of them in an ordered tree-like structure on the hard disk, as we know from MS-DOS (Disk Operating System) for instance. The large branches contain more branches, and the branches at the end contain the tree's leaves or normal files.

Q. Is the directory hierarchy a tree or a network?

3.1.1.2. Sorts of files

Most files are just files, called *regular* files; they contain normal data, for example text files, executable files or programs, input for or output from a program and so on.
While it is reasonably safe to suppose that everything you encounter on a Linux system is a file, there are some exceptions.

- *Directories*: files that are lists of other files.
- *Special files*: the mechanism used for input and output. Most special files are in /dev.
- *Links*: a system to make a file or directory visible in multiple parts of the system's file tree.
- *(Domain) sockets*: a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.
- *Named pipes*: act more or less like sockets and form a way for processes to communicate with each other, without using network socket semantics.

The -l option to **ls** displays the file type, using the first character of each input line:

```
jaime:~/Documents> ls -l
total 80
-rw-rw-r--   1 psdg   faculty   31744 Feb 21 17:56 intro Linux.doc
-rw-rw-r--   1 psdg   faculty   41472 Feb 21 17:56 Linux.doc
drwxrwxr-x   2 psdg   faculty    4096 Feb 25 11:50 itenv
```

This table gives an overview of the characters determining the file type:
Table. File types in a long list

Symbol	Meaning
-	Regular file
d	Directory
l	Link
c	Special file
s	Socket
p	Named pipe
b	Block device

In order not to always have to perform a long listing for seeing the file type, a lot of systems by default don't issue just **ls**, but **ls -F**, which suffixes file names with one of the characters "/=*|@" to indicate the file type. As a user, one only needs to deal directly with plain files, executable files, directories and links. The special file types are there for making your system do what you demand from it and are dealt with by system administrators and programmers. Now, before we look at the important files and directories, we need to know more about partitions.

About partitioning

Why partition?

Most people have a vague knowledge of what partitions are, since every operating system has the ability to create or remove them. It may seem strange that Linux uses more than one partition on the same disk, even when using the standard installation procedure, so some explanation is called for.

One of the goals of having different partitions is to achieve higher data security in case of disaster. By dividing the hard disk in partitions, data can be grouped and separated. When an accident occurs, only the data in the partition that got the hit will be damaged, while the data on the other partitions will most likely survive. This principle dates from the days when Linux didn't have journaled file systems and power failures might have lead to disaster. The use of partitions remains for security and robustness reasons, so a breach on one part of the system doesn't automatically mean that the whole computer is in danger. This is currently the most important reason for partitioning. A simple example: a user creates a script, a program or a web application that starts filling up the disk. If the disk contains only one big partition, the entire system will stop functioning if the disk is full. If the user stores the data on a separate partition, then only that (data) partition will be affected, while the system partitions and possible other data partitions keep functioning.

Having a journaled file system only provides data security in case of power failure and sudden disconnection of storage devices. This does not protect your data against bad blocks and logical errors in the file system. In those cases, you should use a RAID (Redundant Array of Inexpensive Disks) solution.

Partition layout and types

There are two kinds of major partitions on a Linux system:

- *data partition*: normal Linux system data, including the *root partition* containing all the data to start up and run the system; and
- *swap partition*: expansion of the computer's physical memory, extra memory on hard disk.

Most systems contain a root partition, one or more data partitions and one or more swap partitions. Systems in mixed environments may contain partitions for other system data, such as a partition with a FAT or VFAT file system for MS Windows data.

Most Linux systems use **fdisk** at installation time to set the partition type. This usually happens automatically. In some cases, one will need to select the partition type manually and even manually do the actual partitioning. The standard Linux partitions have number 82 for swap and 83 for data, which can be journaled (ext3) or normal (ext2, on older systems). The **fdisk** utility has built-in help, should you forget these values.

Apart from these two, Linux supports a variety of other file system types, such as the relatively new Reiser file system, JFS, NFS, FATxx and many other file systems natively available on other (proprietary) operating systems.

The standard root partition (indicated with a single forward slash, */*) is about 100-500 MB, and contains the system configuration files, most basic commands and server programs, system libraries, some temporary space and the home directory of the administrative user. A standard installation requires about 250 MB for the root partition.

Swap space (indicated with *swap*) is only accessible for the system itself, and is hidden from view during normal operation. Swap is the system that ensures, like on normal UNIX systems, that one can keep on working, whatever

happens. The swap or virtual memory procedure has long been adopted by operating systems outside the UNIX world by now.

Using memory on a hard disk is naturally slower than using the real memory chips of a computer, but having this little extra is a great comfort. Linux generally counts on having twice the amount of physical memory in the form of swap space on the hard disk. When installing a system, you have to know how you are going to do this. An example on a system with 512 MB of RAM:

- 1st possibility: one swap partition of 1 GB
- 2nd possibility: two swap partitions of 512 MB
- 3rd possibility: with two hard disks: 1 partition of 512 MB on each disk.

The last option will give the best results when a lot of I/O is to be expected.

Read the software documentation for specific guidelines. Some applications, such as databases, might require more swap space. Others, such as some handheld systems, might not have any swap at all by lack of a hard disk. Swap space may also depend on your kernel version.

The kernel is on a separate partition as well in many distributions, because it is the most important file of your system. If this is the case, one will find that there is a */boot* partition, holding your kernel(s) and accompanying data files.

The rest of the hard disk(s) is generally divided in data partitions, although it may be that all of the non-system critical data resides on one partition, for example when you perform a standard workstation installation. When non-critical data is separated on different partitions, it usually happens following a set pattern:

- a partition for user programs (*/usr*)
- a partition containing the users' personal data (*/home*)
- a partition to store temporary data like print- and mail-queues (*/var*)
- a partition for third party and extra software (*/opt*)

Once the partitions are made, you can only add more. Changing sizes or properties of existing partitions is possible but not advisable.

The division of hard disks into partitions is determined by the system administrator. On larger systems, he or she may even spread one partition over several hard disks, using the appropriate software. Most distributions allow for standard setups optimized for workstations (average users) and for general server purposes, but also accept customized partitions. During the installation process you can define your own partition layout using either your distribution specific tool, which is usually a straight forward graphical interface, or **fdisk**, a text-based tool for creating partitions and setting their properties.

A workstation or client installation is for use by mainly one and the same person. The selected software for installation reflects this and the stress is on common user packages, such as nice desktop themes, development tools, client programs for E-mail, multimedia software, web and other services. Everything is put together on one large partition, swap space twice the amount of RAM is added and your generic workstation is complete, providing the largest amount of disk space possible for personal use, but with the disadvantage of possible data integrity loss during problem situations.

On a server, system data tends to be separate from user data. Programs that offer services are kept in a different place than the data handled by this service. Different partitions will be created on such systems:

- a partition with all data necessary to boot the machine
- a partition with configuration data and server programs
- one or more partitions containing the server data such as database tables, user mails, an ftp archive etc.
- a partition with user programs and applications
- one or more partitions for the user specific files (home directories)
- one or more swap partitions (virtual memory)

Servers usually have more memory and thus more swap space. Certain server processes, such as databases, may require more swap space than usual; see the specific documentation for detailed information. For better performance, swap is often divided into different swap partitions.

Mount points

All partitions are attached to the system via a mount point. The mount point defines the place of a particular data set in the file system. Usually, all partitions are connected through the *root* partition. On this partition, which is indicated with the slash (/), directories are created. These empty directories will be the starting point of the partitions that are attached to them. An example: given a partition that holds the following directories:

```
videos/                cd-images/          pictures/
```

We want to attach this partition in the filesystem in a directory called /opt/media. In order to do this, the system administrator has to make sure that the directory /opt/media exists on the system. Preferably, it should be an empty directory. Then, using the **mount** command, the administrator can attach the partition to the system. The formerly empty directory /opt/media, it will contain the files and directories that are on the mounted medium (hard disk or partition of a hard disk, CD, DVD, flash card, USB or other storage device).
During system startup, all the partitions are thus mounted, as described in the file /etc/fstab. Some partitions are not mounted by default, for instance if they are not constantly connected to the system, such like the storage used by your digital camera. If well configured, the device will be mounted as soon as the system notices that it is connected, or it can be user-mountable, i.e. you don't need to be system administrator to attach and detach the device to and from the system.
On a running system, information about the partitions and their mount points can be displayed using the **df** command (which stands for *disk full* or *disk free*). In Linux, **df** is the GNU version, and supports the -h or *human readable* option which greatly improves readability. Note that commercial UNIX machines commonly have their own versions of **df** and many other commands. Their behavior is usually the same, though GNU versions of common tools often have more and better features.
The **df** command only displays information about active non-swap partitions. These can include partitions from other networked systems, like in the example below where the home directories are mounted from a file server on the network, a situation often encountered in corporate environments.

```
psdg:~> df -h
Filesystem          Size  Used Avail Use% Mounted on
/dev/hda8           496M  183M  288M  39% /
/dev/hda1           124M  8.4M  109M   8% /boot
/dev/hda5            19G   15G  2.7G  85% /opt
/dev/hda6           7.0G  5.4G  1.2G  81% /usr
/dev/hda7           3.7G  2.7G  867M  77% /var
fs1:/home           8.9G  3.7G  4.7G  44% /.automount/fs1/root/home
```

More file system layout

Visual

For convenience, the Linux file system is usually thought of in a tree structure. On a standard Linux system the following layout generally follows the scheme presented below.
This is a layout from a RedHat system. Depending on the system admin, the operating system and the mission of the UNIX machine, the structure may vary, and directories may be left out or added at will. The names are not even required; they are only a convention.
The tree of the file system starts at the trunk or *slash*, indicated by a forward slash (/). This directory, containing all underlying directories and files, is also called the *root directory* or "the root" of the file system.
Directories that are only one level below the root directory are often preceded by a slash, to indicate their position and prevent confusion with other directories that could have the same name. When starting with a new system, it is always a good idea to take a look in the root directory. Let's see what you could run into:

```
emmy:~> cd /
emmy:/> ls
bin/   dev/  home/    lib/        misc/  opt/     root/  tmp/  var/
boot/  etc/  initrd/  lost+found/  mnt/   proc/    sbin/  usr/
```

Subdirectories of the root directory

Directory	Content
/bin	Common programs, shared by the system, the system administrator and the users.
/boot	The startup files and the kernel, vmlinuz. In some recent distributions also grub data. Grub is the GRand Unified Boot loader and is an attempt to get rid of the many different boot-loaders we know today.
/dev	Contains references to all the CPU peripheral hardware, which are represented as files with special properties.

Directory	Content
/etc	Most important system configuration files are in /etc, this directory contains data similar to those in the Control Panel in Windows
/home	Home directories of the common users.
/initrd	(on some distributions) Information for booting. Do not remove!
/lib	Library files, includes files for all kinds of programs needed by the system and the users.
/lost+found	Every partition has a lost+found in its upper directory. Files that were saved during failures are here.
/misc	For miscellaneous purposes.
/mnt	Standard mount point for external file systems, e.g. a CD-ROM or a digital camera.
/net	Standard mount point for entire remote file systems
/opt	Typically contains extra and third party software.
/proc	A virtual file system containing information about system resources. More information about the meaning of the files in proc is obtained by entering the command **man *proc*** in a terminal window. The file proc.txt discusses the virtual file system in detail.
/root	The administrative user's home directory. Mind the difference between /, the root directory and /root, the home directory of the *root* user.
/sbin	Programs for use by the system and the system administrator.
/tmp	Temporary space for use by the system, cleaned upon reboot, so don't use this for saving any work!
/usr	Programs, libraries, documentation etc. for all user-related programs.
/var	Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it.

How can you find out which partition a directory is on? Using the **df** command with a dot (.) as an option shows the partition the current directory belongs to, and informs about the amount of space used on this partition:

```
sandra:/lib> df -h .
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda7             980M  163M  767M  18% /
```

As a general rule, every directory under the root directory is on the root partition, unless it has a separate entry in the full listing from **df** (or **df -h** with no other options).

The file system in reality

For most users and for most common system administration tasks, it is enough to accept that files and directories are ordered in a tree-like structure. The computer, however, doesn't understand a thing about trees or tree-structures. Every partition has its own file system. By imagining all those file systems together, we can form an idea of the tree-structure of the entire system, but it is not as simple as that. In a file system, a file is represented by an *inode*, a kind of serial number containing information about the actual data that makes up the file: to whom this file belongs, and where is it located on the hard disk.

Every partition has its own set of inodes; throughout a system with multiple partitions, files with the same inode number can exist.

Each inode describes a data structure on the hard disk, storing the properties of a file, including the physical location of the file data. When a hard disk is initialized to accept data storage, usually during the initial system installation process or when adding extra disks to an existing system, a fixed number of inodes per partition is created. This number will be the maximum amount of files, of all types (including directories, special files, links etc.) that can exist at the same time on the partition. We typically count on having 1 inode per 2 to 8 kilobytes of storage.

At the time a new file is created, it gets a free inode. In that inode is the following information:

- Owner and group owner of the file.
- File type (regular, directory, ...)
- Permissions on the file
- Date and time of creation, last read and change.

- Date and time this information has been changed in the inode.
- Number of links to this file
- File size
- An address defining the actual location of the file data.

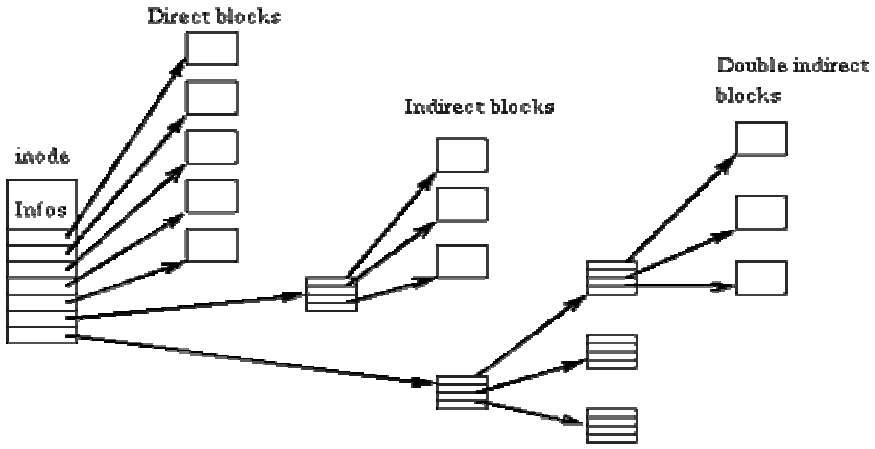
The only information not included in an inode, is the file name and directory. These are stored in the special directory files. By comparing file names and inode numbers, the system can make up a tree-structure that the user understands. Users can display inode numbers using the -i option to ls. The inodes have their own separate space on the disk.

Basic File System Concepts

Every Linux filesystem implements a basic set of common concepts derivated from the Unix operating system files are represented by inodes, directories are simply files containing a list of entries and devices can be accessed by requesting I/O on special files.

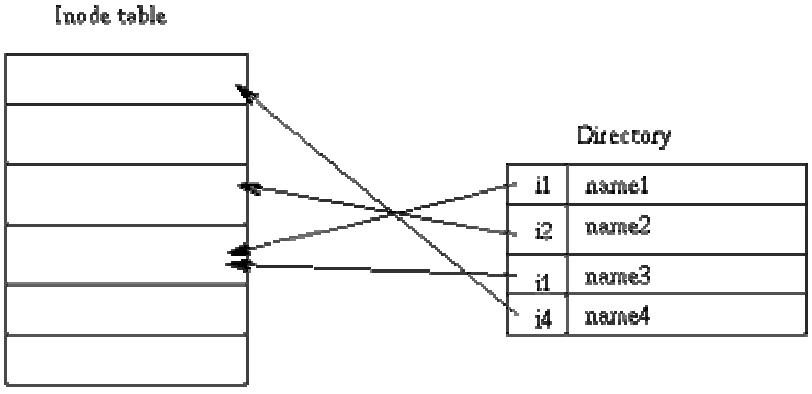
Inodes

Each file is represented by a structure, called an inode. Each inode contains the description of the file: file type, access rights, owners, timestamps, size, pointers to data blocks. The addresses of data blocks allocated to a file are stored in its inode. When a user requests an I/O operation on the file, the kernel code converts the current offset to a block number, uses this number as an index in the block addresses table and reads or writes the physical block. This figure represents the structure of an inode:



Directories

Directories are structured in a hierarchical tree. Each directory can contain files and subdirectories. Directories are implemented as a special type of files. Actually, a directory is a file containing a list of entries. Each entry contains an inode number and a file name. When a process uses a pathname, the kernel code searchs in the directories to find the corresponding inode number. After the name has been converted to an inode number, the inode is loaded into memory and is used by subsequent requests.



This figure represents a directory:

Links

Unix filesystems implement the concept of link. Several names can be associated with a inode. The inode contains a field containing the number associated with the file. Adding a link simply consists in creating a directory entry,

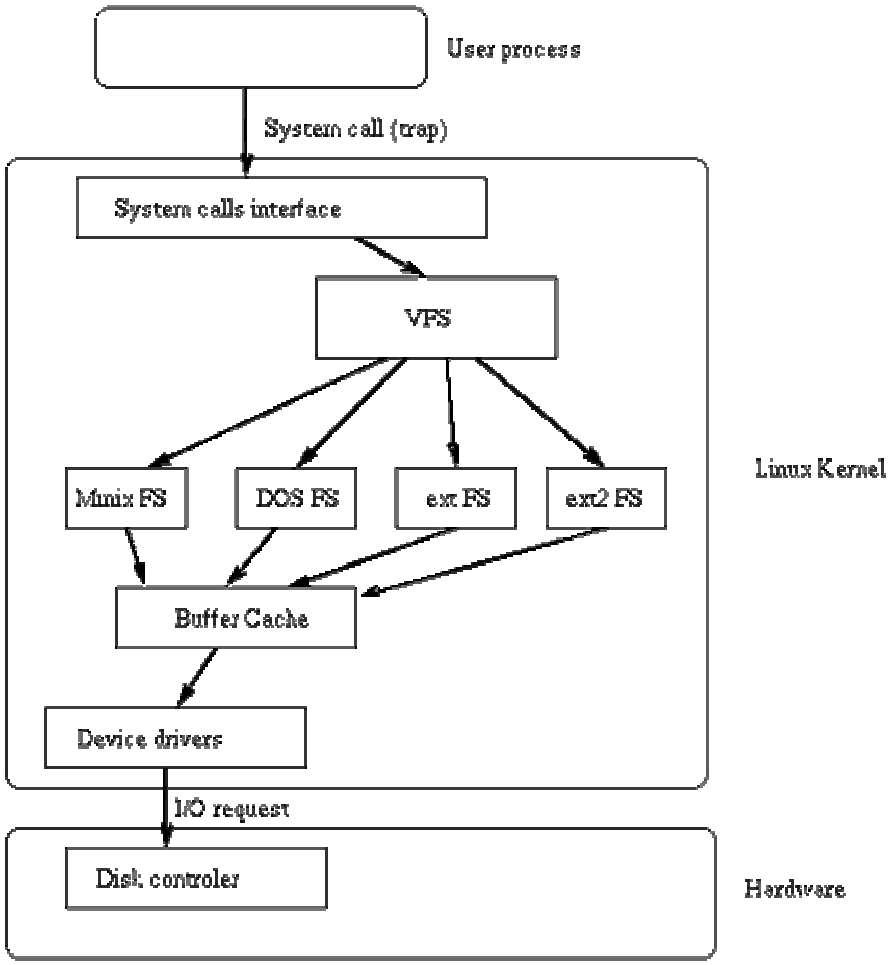
where the inode number points to the inode, and in incrementing the links count in the inode. When a link is deleted, i.e. when one uses the rm command to remove a filename, the kernel decrements the links count and deallocates the inode if this count becomes zero.
This type of link is called a hard link and can only be used within a single filesystem: it is impossible to create cross-filesystem hard links. Moreover, hard links can only point on files: a directory hard link cannot be created to prevent the apparition of a cycle in the directory tree.
Another kind of links exists in most Unix filesystems. Symbolic links are simply files which contain a filename. When the kernel encounters a symbolic link during a pathname to inode conversion, it replaces the name of the link by its contents, i.e. the name of the target file, and restarts the pathname interpretation. Since a symbolic link does not point to an inode, it is possible to create cross-filesystems symbolic links. Symbolic links can point to any type of file, even on nonexistent files. Symbolic links are very useful because they don't have the limitations associated to hard links. However, they use some disk space, allocated for their inode and their data blocks, and cause an overhead in the pathname to inode conversion because the kernel has to restart the name interpretation when it encounters a symbolic link.

Q. What is the minimum link count of a directory in Linux/Unix?

Device special files
In Unix-like operating systems, devices can be accessed via special files. A device special file does not use any space on the filesystem. It is only an access point to the device driver. Two types of special files exist: character and block special files. The former allows I/O operations in character mode while the later requires data to be written in block mode via the buffer cache functions. When an I/O request is made on a special file, it is forwarded to a (pseudo) device driver. A special file is referenced by a major number, which identifies the device type, and a minor number, which identifies the unit.

The Virtual File System
Principle
The Linux kernel contains a Virtual File System layer which is used during system calls acting on files. The VFS is an indirection layer which handles the file oriented system calls and calls the necessary functions in the physical filesystem code to do the I/O.
This indirection mechanism is frequently used in Unix-like operating systems to ease the integration and the use of several filesystem types.
When a process issues a file oriented system call, the kernel calls a function contained in the VFS. This function handles the structure independent manipulations and redirects the call to a function contained in the physical filesystem code, which is responsible for handling the structure dependent operations. Filesystem code uses the

buffer cache functions to request I/O on devices. This scheme is illustrated in this figure:



The VFS structure

The VFS defines a set of functions that every filesystem has to implement. This interface is made up of a set of operations associated to three kinds of objects: filesystems, inodes, and open files.

The VFS knows about filesystem types supported in the kernel. It uses a table defined during the kernel configuration. Each entry in this table describes a filesystem type: it contains the name of the filesystem type and a pointer on a function called during the mount operation. When a filesystem is to be mounted, the appropriate mount function is called. This function is responsible for reading the superblock from the disk, initializing its internal variables, and returning a mounted filesystem descriptor to the VFS. After the filesystem is mounted, the VFS functions can use this descriptor to access the physical filesystem routines.

A mounted filesystem descriptor contains several kinds of data: informations that are common to every filesystem types, pointers to functions provided by the physical filesystem kernel code, and private data maintained by the physical filesystem code. The function pointers contained in the filesystem descriptors allow the VFS to access the filesystem internal routines.

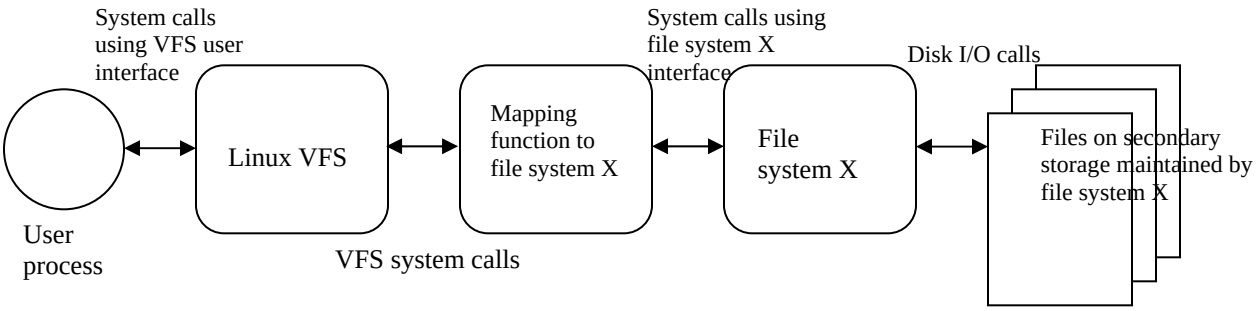
Two other types of descriptors are used by the VFS: an inode descriptor and an open file descriptor. Each descriptor contains informations related to files in use and a set of operations provided by the physical filesystem code. While the inode descriptor contains pointers to functions that can be used to act on any file (e.g. create, unlink), the file descriptors contains pointer to functions which can only act on open files (e.g. read, write).

The Second Extended File System

Motivations

The Second Extended File System has been designed and implemented to fix some problems present in the first Extended File System. Our goal was to provide a powerful filesystem, which implements Unix file semantics and offers advanced features.

Of course, we wanted to Ext2fs to have excellent performance. We also wanted to provide a very robust filesystem in order to reduce the risk of data loss in intensive use. Last, but not least, Ext2fs had to include provision for extensions to allow users to benefit from new features without reformatting their filesystem



- In the VFS, a file may be created, read from, written to, or deleted. For any file system, a mapping module is needed to transform the characteristics of the real file system to the characteristics expected by the VFS.
- The figure above shows the key ingredients of the Linux file system strategy. A user process issues a file system call (e.g., OPEN) using the VFS. VFS then converts this to an internal (kernel) file system call, that is passed to a mapping function for a specific file system (e.g., NTFS).
- The target file system software is then invoked to perform the requested function on a file or a directory under its control and secondary storage.
- Results of the operation are then communicated back to the user in a similar way.