4 Files

Persistent file stores allow for more data than will fit into the address space of a process; for data to survive the termination of the process; and for several processes to access the data concurrently. It must be possible to create and delete files, access them for reading and writing, identify them by symbolic names; prevent accidental sharing but permit deliberate sharing; manage secondary storage space to contain the files; and in particular protect files against system failures. The problems are very much the same as those of memory management, but with the additional cost of ensuring that the files are undamaged even by the failure of their (current) owner.

In UNIX a file is just a byte stream. Any higher-level organisation is a user-level problem. In consequence pretty much every program can operate unchanged on pretty much any file, wherever it happens to be stored. Other systems might provide special files, perhaps indexed for random access; so it is not necessary for every user program to implement its own indexing.

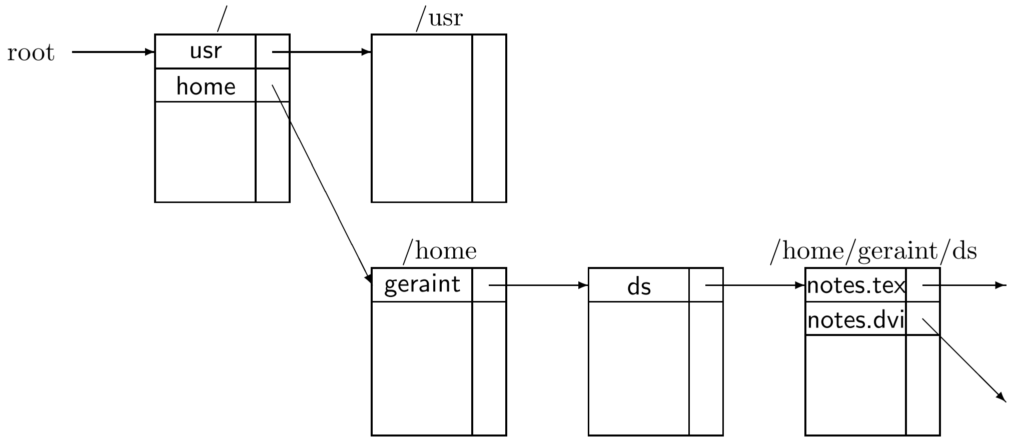
4.1 Directories

Directories provide device independence by mapping file names to files. This independence is not very well implemented in MS-DOS! The simplest of directories (e.g. in CP/M) are tables, one per disk, mapping a file name to a location on the device, a file size, perhaps some indication of type, access control information, and other administrative administration such as time of access.

As file systems get larger, there is a need for a tree (not usually a general graph) of directories. Directories are files, and one directory is identified as the root. MS-DOS gets this from UNIX, UNIX got it from MULTICS. File names are now paths from the root of the tree to a file.

In UNIX the directory tree is an index for files; the files are separate. Each directory maps a name (the last component of the path name) to the identity of a file; nothing else appears in the directory. A file can appear several times in the directory tree. The UNIX implementation of this idea probably comes from the contemporary OSPub.[5]

[5] Joe Stoy and Christopher Strachey, *OS6 – an experimental operating system for a small computer*, Comp. J. 15, No8. 2 and 3, 1972. Strachey (1916–1975) was at the time the only Oxford academic computer scientist, and Joe Stoy was his research officer; they and a handful of students were the computer science department.



Directory entries contain i-numbers, which are indexes to a table of i-nodes[6] on the disk alongside the data in the files. Each i-node contains the properties of a file which in other systems would have been in the directory entry, including the location of its contents.

UNIX directories are marked as being of a special type, essentially so that it is possible to identify the directory tree as distinct from the files entered in it. This allows file system maintenance utilities to do things like visiting every file. Apart from that they are ordinary files.

Creating a (temporary) file amounts to creating an i-node for it, but giving it a name and making it persistent involved entering it into a directory. The two are logically distinct operations. In particular a file can be re-entered under another name, or in another directory, by the system call *link*. The two entries created are entirely equivalent, the file is the same one, there is no subsequent sense of the 'original' entry being special.

4.2 Special entries

Every directory in a UNIX file system has an entry for itself, with name '.', and its parent in the tree with name '..'. (By convention the root of the filesystem is its own parent.) Since they are essential for the integrity of the file system, making and changing these special entries is a privileged operation, and they are normally created by the system call that makes a new directory.

A single file system in UNIX has to be stored on a single block device, but the files of an operating system be spread over several file systems. The root of a filsystem can be substituted for a directory anywhere in another file system. This is called *mounting* one file system on a *mount point* in the other. In that case the

[6]Nobody knows why they are called *i-nodes*, though it seems likely that they are *index nodes*; the corresponding structure in OSPub was called a *header*.

'`..`' entry of the root of the mounted file system is made to appear to refer to the parent of the mount point.

A convention has grown up that files with names that begin with a dot are used for 'internal workings' of programs and operating systems, rather thyat the user's own files. So it is conventional that they do not appear in directory listsings. (`ls` does not show them, but `ls -a` does.) There is however nothing special about them, except for '`.`' and' `..`'.

4.3 File contents

Contiguous files would be easy to describe, and are fast to access but suffer from disk fragmentation. They are fine for read-only file systems (like big stable databases and audio CDs).

In CP/M the directory contained up to 16 block numbers; bigger files have a second fake directory entry. In MS-DOS the directory points at the first block, subsequent blocks are in a *file allocation table* which is a mapping from current block to next block number for the whole disk. In OSPub each block contained the number of the next block (and the previous block).

UNIX files are more deeply structured, in the way outlined in figure 1. Each i-node contains the number of the first few (ten) blocks of the file. If that is not enough it contains the number of a *single indirect block* which contains the numbers of the next n blocks. The number n is determined by the block size and the number of bytes in a block number, which in turn depends on the size of the space occupied by the file system. If that is not enough it contains the number of a *double indirect block* which contains the numbers of up to n further single indirect blocks. If that is not enough, it contains the number of a *triple indirect block* which contains the numbers of up to n further double indirect blocks. This makes random access to the file reasonably quick, whilst making small files very efficient.

4.4 Disk space management

The problems of managing space in a disk file system are essentially similar to those of memory management. Disk file system space comes in blocks because disks are efficiently accessed in blocks, and so that disk addresses can be efficient. However the size of blocks is (as always) a compromise. Blocks that are too big suffer from internal fragmentation.

When is a block free to be allocated for use in a file? Essentially when it is not in a file already. This *can* be determined by walking over the directory tree; however that is both inefficient and difficult to do atomically. For efficiency one keeps a redundant representation of the free space: for example a file containing

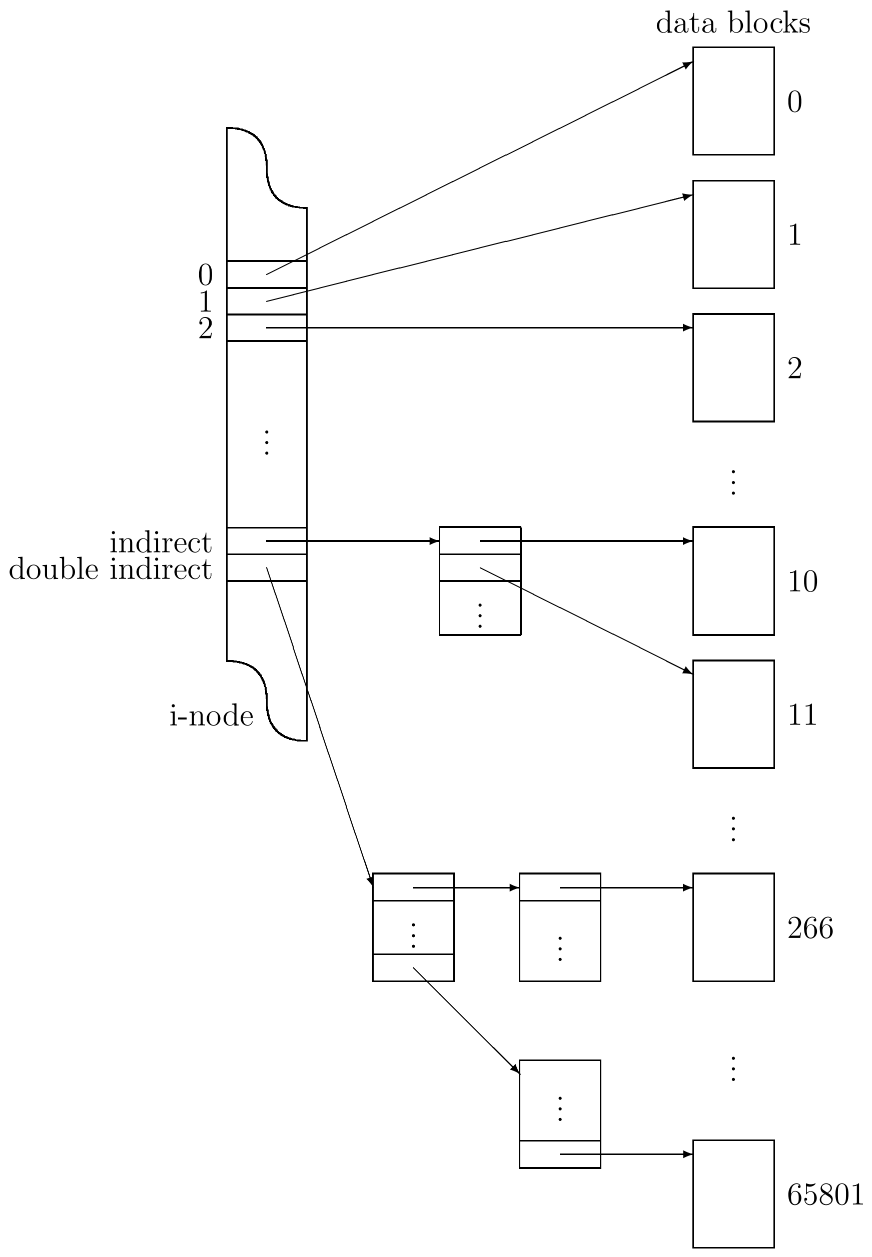


Figure 1: Outline of the structure of a UNIX file.

all the free blocks; or a file listing all the free block numbers; or a bitmap of the blocks on the disk showing which are free.

The existence of a redundant representation makes it possible (and necessary) to check for compatibility between reality and the free list. For example, after crash it is possible that the free list is wrong because several file system operations will be non-atomic. Deleting a file, for example, must remove it from the file system and subsequently release the freed space. Each of these will involve disk writes. A crash between these two will cause a *space leak*.

A file system check will walk over the structure of the directory tree and of all of the files in the tree, establishing the consistency of the tree and the structures of the files. In the course of doing so it can establish which i-nodes and which blocks are actually in use, and so construct accurate free lists. This is a time-consuming task: UNIX used to do call `fsck` whenever it was booted. As an optimisation, it would only happen if the file system was not marked as 'clean', or if it had not been checked for some long time. However, as file systems grow larger it can take quite substantial times to run a file system check a large system.

Modern file systems usually incorporate *journalling* to speed up recovery. At the expense of potentially doubling disk write traffic, the journal records the sequence of disk operations which make up a transaction. The real transaction is not written until the journal has been written. After a crash it is possible to restore the file system to a consistent state by running through the last few journalled disk writes.

4.5 UNIX files

Most programs in UNIX reading a file would use calls to library routines like `fopen` and `fread` which manage reads from the file through a buffer in the process itself. These library routines call system calls, the essential ones of which are

`fd = open("/home/geraint/ds/notes.tex", O_READ)`
returns a small integer *fd* called the *file descriptor*;

`nread = read(fd, buffer, n)`
reads up to *b* bytes from the described file;

`close(fd)`
closes the file and makes the descriptor invalid.

Each process contains a file descriptor table that maps file descriptors to *file pointers*. The file pointer table contains, for each open file, a current offset in the file and an *i-number*. The i-node cache maps i-numbers to i-nodes that contain block numbers. Why so many levels of indirection? The idea is to manage the interac-

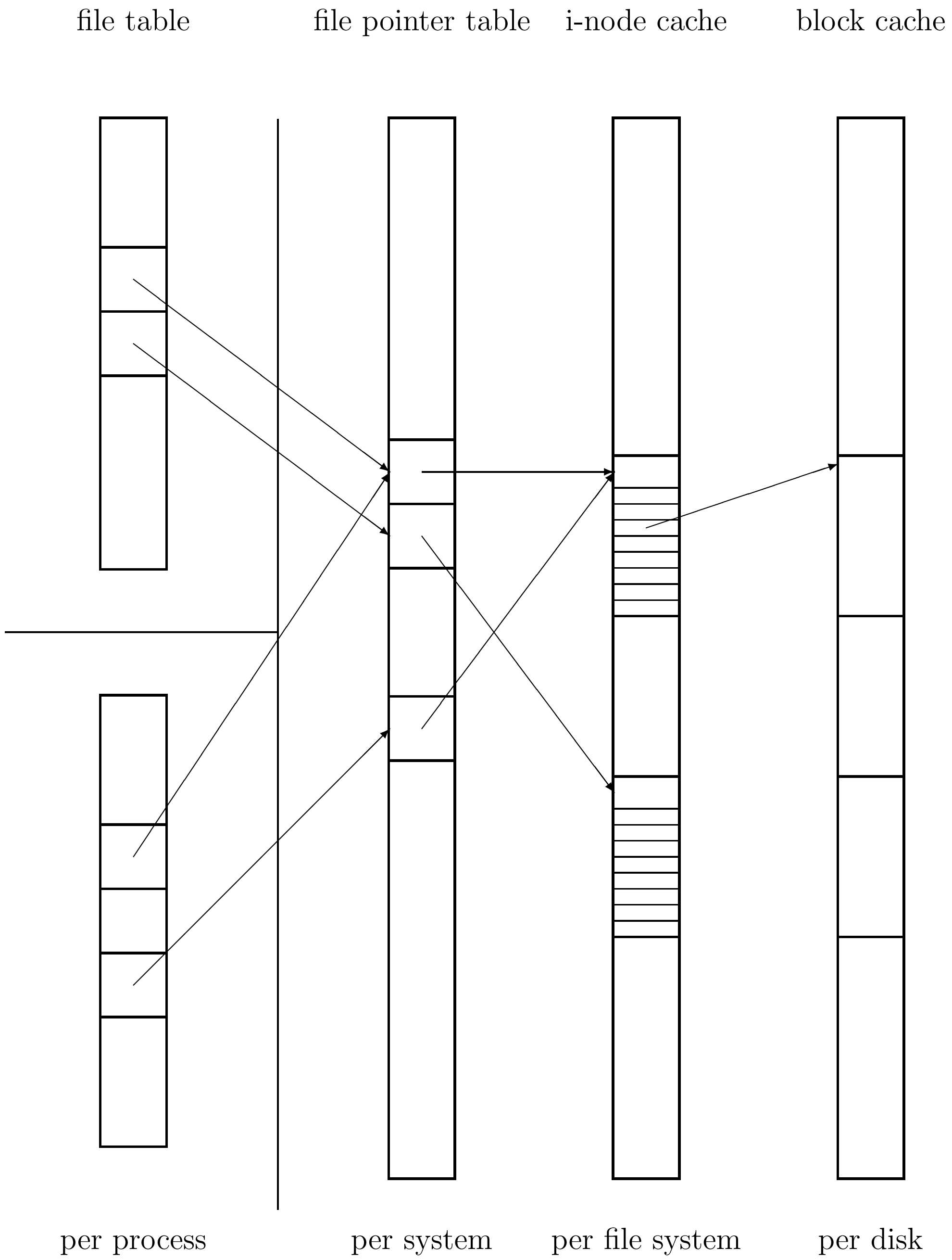


Figure 2: The data structures corresponding to open files.

tion of several processes reading or writing the same file. If two processes write a file in sequence, for example in the shell script

```
( echo "The end is nigh" > doom );
( date > doom )
```

the shell forks two process in sequence. The first opens the file *doom* for writing, truncating it to length zero and then writes the slogan; when that terminates and closes the file, the second opens the file again, again truncating it to empty, and writes the date. When it terminates and closes the file it contains only the date. Conversely

```
( echo "The end is nigh" ; date ) > doom
```

opens the file in the shell itself, truncating it. It then forks a new process which is given a copy of the file descriptor table containing the descriptor of the open file. This process then forks in turn processes which write the slogan, and then the date, to that same file. And finally the file is closed.

Entries in the i-node cache correspond to tapes, and those in the the file-pointer table to read or write heads at a given position on the tape. The file descriptor table identifies which of these heads corresponds to a file identified by its function in the program. Descriptors that refer to the same file pointer can be traced back to the same call of *open*, and reading or writing one will move the position of the next read or write of the other. Descriptors that refer to different file pointers but the same i-number derive from two different calls of *open* and reads are independent (but writes will conflict, of course).

4.6 Write-back caching

The block cache reduces disk traffic: it will tend to hold the blocks of frequently read files, such as perhaps the root directory and the current directory. Additionally UNIX makes this a *write-back* cache. That is, when file is changed the block in the cache is written, and the block is marked as dirty, but the write to the disk does not happen until later. This creates an obligation to write back any dirty blocks before they are evicted from the cache.

The system call *sync* writes back all the dirty blocks from the block cache. Until this happens there is a high probability that the disk contains an inconsistent image of the file system. Most UNIX systems will have a process running whose sole task is to call *sync* every few seconds.

In contrast, in some other systems the block cache would be *write-through*: disk writes would always go directly to disk as well as updating the cache. This results in many more disk accesses, but a smaller window of vulnerability to crashes. Latterly even Microsoft systems use write-back for hard disks.

4.7 When is a UNIX file deleted?

When a file is deleted, its space becomes free and should be entered in the free list. When does this happen? A file is not deleted until all links to it from the directory structure have been removed. Again, this *could* be determined by walking over the directory tree, but each i-node contains a *reference count*.

Additionally, if a file is currently open its space must not be freed! In particular the file containing the code of a currently executing process must be kept in existence. Accordingly the reference count of the i-node must be increased by the number of open file pointers which refer to it. Only when this reference count becomes zero is the file deleted. Similarly each file pointer may be shared by several processes, and needs to keep a reference count.

4.8 Permissions

One of the more important services of a file system is to protect files against accidental or malicious access, whilst simultaneously permitting legitimate access. Permissions have to be granted to users, so we need some sense of the identity of a user. UNIX identifies users by two small numbers (the user identifier, *uid*, and group identifier, *gid*) which can only be changed by privileged system calls that (conventionally) demand a password. Again, UNIX offers separate read, write and execute permissions, with some extra conventions about what these mean for a directory. Read permission to a directory allows one to see which file names appear in it; execute permission for a directory allows one to look up an entry; and these are independently controlled forms of access.

In general, the rights of access to a system of files which are granted to a set of users constitute a matrix: is user u permitted access to file f?

One style of representing this matrix is the *access control list*, representing a file-wise column of the matrix. An access control list attached to a file f gives an algorithm for determining whether a given user has permission. The list may be simply a list of users with access. Or it may be a list of classes of users paired with whether they have access or not; permission is granted according to rights of the first set containing the user.

One of the fields of a UNIX i-node contains the file access permission bits, which to a first approximation are a nine-bit representation of an access control list; the i-node also identifies an owner and a group. The nine bits represent read, write and execute rights for each of three classes of user: those with *uid* matching the *owner*, those entitled to have a *gid* matching the *group*, and *others*.

In physically distributed systems, this scheme is open to abuse. The meaning of a *uid* on one machine may not match its meaning on another. The ability to restrict changes to *uid* on one machine may not be enforceable on another. This makes the restrictions indicated by access control lists difficult to enforce when a file system on one machine is accessed by process running on another.

An alternative representation of the rights matrix is *capabilities*, which are properties of a file that are granted to users, so that a user's rights depend not on his identity by on list of cababilities which he holds. These sets of cababilities form a user-wise row of the rights matrix. A capability is an essentially unforgeable token (a cryptographically secure bit string) created by the filesystem which must be handed over by a user when requesting access to a file.

4.9 Special files

The byte stream interface provided by a file descriptor makes it easy to provide access to other byte-stream like things through the same interface. The system call *pipe* allocates a buffer and creates two file descriptors, one of which writes into a buffer and the other of which reads from it. By calling *pipe* before a call to *fork* it is possible to connect two processes together in a pipeline.

UNIX also exploits the hierarchical structure of file names to provide access to device drivers. Special i-nodes, usually but not necessarily those entered in `/dev`, carry parameters that refer to device drivers and device numbers. This makes it easy to parametrise a program by which device it uses.

Many modern versions of UNIX provide a file system whose i-nodes give access not to disk space but to kernel data structures. In Linux this is usually `/proc`, which presents as a directory containing various files which when read show the state of the system. In particular there will be a subdirectory for each current process, so that reading `/proc/11803/cmdline` will return the command line of process number 11803. This scheme has the advantage over providing system calls to read every structure that any utility that operates on files can operate on the contents of one of these reports on the state of the system.

Exercises

4.1 Each UNIX i-node contains a link-count, which is a count of the number of times the i-node number appears in a directory entry. This information is redundant: why is it there?

4.2 One of the observations made (on page 35) about the redundant representation of the UNIX file system is that it is possible (and periodically necessary) to check that redundant information is correct.

Compare the file system with the memory management of 'heap' storage used to keep non-stack variables in programming languages like Oberon and Java, or the implementations of languages like HASKELL. Similar efficiency concerns mean that there is probably a redundant representation of the free store. Why is there no equivalent to the file system check (`fsck`)?

4.3 Describe the succession of disk accesses involved in opening and reading a file with a given name, such as `/home/geraint/ds/notes.tex`. Roughly how many disk accesses would you expect this to take?

4.4 It is possible to enquire of a file descriptor which i-number (if any) to which it relates. Given the i-number of the current directory, explain how to find the name of the current directory.

Suppose a process is reading from a certain file descriptor attached to a disk file. Why is it not sensible to ask for the name of the file which it is reading?

4.5 The system call *fork* creates a new process with a file descriptor table that contains a copy of all of the open file descriptors in the parent. What does *fork* have to do to other structures relating to file descriptors?

4.6 Most UNIX file systems allow directories to contain *symbolic links*. These are entries which map a name not to an i-number but to a replacement (path) name. If when you look up a file name in a directory tree you find a symbolic link, the result should be the result of looking up the replacement name.

What differences are there between the 'hard' links described on page 32 and symbolic links? What are the advantages of symbolic links, and their disadvantages?

4.7 OSPub allowed there to be a special entry, *predecessor*, in any directory.[7] The predecessor was logically (if not physically) the last entry, and was always a directory. If you looked up a name in a directory and that name was not there, but there was a predecessor entry, the lookup continued in that predecessor directory.

A new version of some project could be made by creating a new empty directory, and entering the directory of the current version as its predecessor. Any changes that were to be made to files could be made to new versions of those files in the new directory, but other files could be shared unchanged.

What would you do in UNIX to get the same effect, without having to make a copy of all of the files of the project? What are the differences between the two schemes?

[7]Actually it was called PREDECESSOR INDEX, because OSPub directories were called indexes, the keys in directories were *pairs* of strings, and OSPub had a character set with underlined characters in it.

4.8 UNIX comes from an environment in which mini-computers had fixed hard disks, and DOS was built for microcomputers with removable floppy disks. Floppy disks were comparatively slow, and the penalty for a disk write on a floppy would be much greater than that for a write to a hard disk.

Why might it be that DOS block caches were write-through but UNIX has always used write-back?

4.9 Suppose that a certain UNIX file system occupies a terabyte divided into one kilobyte blocks, with ten direct blocks mentioned in the i-nodes, and single, double and triple indirect blocks. Exactly how big a file could you have in this system? How many bits would you need for the file-length entry in the i-node?

Is this big enough for practical purposes? What would you do if it were not? What about bigger file systems?

4.10 How much disk space would be consumed by the FAT (of a MS-DOS style file system) on a disk with blocks like those mentioned in question 4.9? How does this compare with the amount of disk consumed by the indexing in the UNIX file system?

How would your answer be affected if there were more or less than ten direct block pointers in the i-node?

4.11 The bulk storage of the 'hierarchical file store'[8] in OUCS consists of a large number of cassette-mounted magnetic tapes, a smaller number of tape drives and a robot that can carry tapes between the drives and some racks.

Suggest a likely design for the computer system which makes the library of tapes appear to be a large (but very slow) file store. Do any parts of your design require special hardware support?

4.12 Three different protection mechanisms that were discussed are capabilities, access control lists, and the UNIX *rwx* bits. For each of the following protection problems, tell which of these mechanisms can be used.

1. Dave wants his files readable by everyone except Nick.
2. Dave and Nick want to share some secret files.
3. Dave wants some of his files to be public.

The force of this question is in the meaning of *everyone except*. You can assume that everyone who is granted access to a file will co-operate in keeping secrets which are shared with them.

[8] `http://www.oucs.ox.ac.uk/hfs/`

4.13 Suppose a file's protection bits are `------r--` (so the owner has no permissions, but 'others' have read permission). Find out whether the owner can read the file. Deduce the order in which file permissions are checked.