NOTES ON FILE SYSTEMS†

A file system is a service provided by the operating system (OS). Minimally, such a system allows users to *create*, *read*, *write* and *delete* files. Usually OS's support more operations on files, such as *editing*, *copying*, *renaming* files etc.

First we'll discuss file systems in general, without reference to any particular OS. Later we'll look in more detail into the Unix‡ file system.

In our discussion we'll assume (as modem OS's do) that files are stored on disk (or similar storage device), rather than cm tape.

Disk space is organised into cylinders (all tracks with the same radius from the center). A cylinder is comprised of a number of surfaces (two for each platter on the disk). Each track of a cylinder is further subdivided into blocks. Individual blocks are the units of transfer between main memory and secondary storage. Thus, disk address space may be viewed as a 3-dimensional array, where item (i, j, A) is the kth block on the jth surface of the ith cylinder.

File systems abstract this structure of disk address space and view it as a long linear (l-dimensional) array of blocks. If each cylinder has s surfaces and each track has b blocks, then block (i, j, k) in the 3-dimensional view of the disk, is taken to be the $(k + j \cdot b + i \cdot b \cdot s)$th entry in the one-dimensional view of the disk. In effect, the linear array consists of all the blocks of the first cylinder (i.e., all the blocks the first track of the first surface, then the first track of the second surface, $\cdots$, then the first track of the last surface); following are the blocks of the second cylinder (in the same order) and so on. Notice that blocks within the same track and tracks comprising the same cylinder are contiguous in this array.

The one dimensional view of disk space is convenient because it simplifies the space management problem.

SPACE MANAGEMENT

The file system must somehow manage the linear array of blocks. In particular it must:

- Keep track of the blocks not used by any file (the set of such blocks is called the *free* pool, or *free list*).
- Allocate space for a file, when a user creates that file.
- Return to the free pool the space allocated to a file, when that file is deleted.

There are three principal methods of space allocation: segmented allocation, linked allocation and indexed allocation. Before describing these, let's first consider the question of how to keep track of the free pool.

Managing the free pool

There are several techniques for keeping track of the disk blocks that are free. The main are:

1. *Linked list of free blocks:* All free blocks are linked together to form a list. A special location is used to store a pointer to the first block in that list. We may also keep there the length of the list, so that if a user requests k blocks for a file, the file system can tell whether it can satisfy the request without having to traverse the entire list. (Note that traversing the list is time consuming since moving from one block to the next requires a disk access.)

2. *Linked list* of **segments:** A *segment* is a set of contiguous blocks (i.e. blocks that are successive in the linear array). A free *segment* is a segment consisting entirely of free blocks. Instead of linking together individual free blocks we link together entire free segments. In each segment we store its *size* (the number of blocks it consists of) and a pointer to the next free segment. In addition, a pointer to the first segment in this list is kept in a special location, (This method is used in conjunction with the segmented allocation technique, discussed below.)

† Notes by V. Hadzilacos.

‡ Unix is a trademark of AT&T Bell Laboratories.

3. *Bitmap*: We keep an array of as many bits as there are blocks. Bit i is off if and only if block i is free.

Segmented allocation

In this allocation technique, the free list is managed by using the linked list of segments (*cf.* 2 above). Initially (when all blocks are free) the free list consists of a single segment that encompasses all blocks.

When a file is created, a segment consisting of some fixed number of blocks is allocated for it. Thus, the tie creation request must specify the size of the file to be created, so that the needed number of blocks can be allocated

Suppose a user has requested the creation of a k-block file. The file system scans the free list to find some segment S of at least k blocks. It then removes S from the free list and breaks it into two pieces: S_1, of k blocks, is allocated for the user's file; the remainder, S_2, is returned to the free list.

An interesting question is, how to choose S in the event the free list contains several segments of size at least k. There are several policies one might use. Most popular are:

1. *First fit:* Choose the first segment S of size at least k in the free list.
2. *Best fit:* Choose the smallest segment S of size at least k in the free list.

There are other possibilities too. No known strategy is superior to all others in all cases. As a matter of practical interest, first fit is the one most commonly used. (Among other advantages, note that it does not require the system to scan the entire free list: we can stop as soon as the first "big enough" segment is found. On the contrary, to determine the block that "best fits" a request, the entire free list must be scanned.)

When a file is deleted, its allocated segment is return4 to the list of free segments. If one or both adjacent segments are also free, the file *system* will consolidate them into a single, "large" free segment.

Advantages:

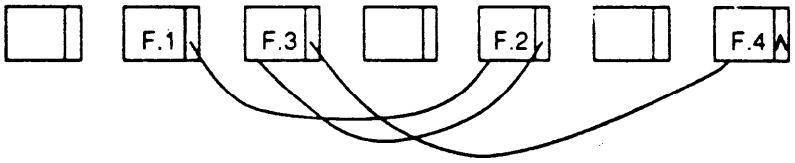
1. Maintains physical contiguity of blocks belonging to the same file, thereby minimising time spent on seeks.
2. It is very easy to directly access the ith block of a file. Thus, this allocation policy is suitable for direct file access.

Disadvantages:

1. Very inflexible: Can't increase 'the size of a file dynamically.
2. Space is wasted for two reasons: (a) the maximum number of blocks a file might need must be allocated, even if that maximum size has not been reached yet; and (b) external fragmentation (there may be enough space to fit a new record, but none of the free segments by itself is big enough).

Linked allocation

The blocks of a file are linked together. Successive blocks of the file are not necessarily physically contiguous. This is illustrated in the diagram below.



Advantages:

1. No external fragmentation problems.
2. Size of a file can change dynamically.
3. Can modify a file by writing only the affected blocks (no need to copy the entire file).

Disadvantages:

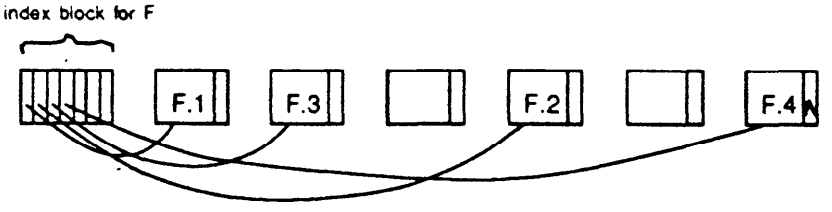
1. Can't directly access the ***ith*** block of a file (the previous blocks *must* be accessed sequentially). Thus this allocation method is not useful for direct files.

2. Sequential file processing may require wild seeks, since blocks of the same file are not (necessarily) physically contiguous.
3. Space must be reserved for pointers in each block If the block size is small then the fraction of space devoted to pointers could be significant

Indexed **allocation**

For each file we keep an index of the blocks used by that file. The index is a list of pointers, the ***ith*** of which points to the *i*th block in the file. Thus, instead of block pointers being embedded in the blocks of the file (as in the linked allocation method), the pointers are all kept together, in one place (the index).

The index itself is kept as a separate block, the ***index block. The*** system keep one such block for each file. This technique is illustrated by the following diagram.



So, to find where the *i*th block of file ***F*** is, we simply look up the *i*th pointer in ***F's*** index block In particular, it is not necessary to scan the i first blocks of a file, as is the case in the linked allocation method

A natural question is: what do we do if the number of blocks in a file exceeds the number of pointers that may be kept in a block? The solution is to have a multi-level index. That is the index blocks **contains** pointers to blocks that have pointers to blocks . . . that have pointers to the file's data blocks. As it turns out, a **3-level** index (root and two more levels) is sufficient for **most** cases. Unix uses a similar method, to be described in more detail **later** on.

Advantages:

1. No external fragmentation.
2. The file may be mcxlified by changing the index and the affected data blocks (no **need to copy the** entire file.)
3. It is relatively easy to locate the *i*th block of a file. In particular this can be done without sequentially accessing all previous blocks. Thus this method can be used for direct access.

Disadvantages:

1. Space must be devoted to the index block(s).
2. Physical contiguity of the blocks in the same file is not guaranteed.
3. Compared to segmented allocation, finding the *i*th block of a file is more time consuming. *(A* simple calculation is sufficient in segmented allocation; accessing index blocks is required in indexed allocation.)

FILE DIRECTORIES

A file system keeps track of its files and other relevant information in a structure called **a *file directory*.** Each file is described by a *directory **entry*** that contains information such as:

- The symbolic name of the file (the name given by the user).
- Information on how to locate the file. This depends on the allocation technique. For instance, in segmented . allocation or linked allocation, a pointer to the first block of the file **may** be given; in indexed allocation **a** pointer to the (root) index block may be supplied
- The size of the file.
- Access information: Who and when last read/wrote the file?

- Ownership and access privileges: Who owns the file? Who can read/write the file?

The tile directory must be kept on disk because it is too large and because it must survive crashes, system shutdowns etc. Thus, the directory itself must be kept in one or more files.

On the directory file(s) we want to perform the following operations: insert an entry (when a file is created); delete an entry (when a file is deleted); modify an entry (when a file is accessed, its access privileges are changed etc.); and display all entries in the directory.

It is imperative to be able to perform these operations extremely efficiently because the directory entry for a file must be consulted and/or modified for every single operation on that file. The desired efficiency is usually achieved in the following manner:

- The system maintains an Active *File Table* (AFT) in main memory. When a user opens a file *F, the* file system finds the directory entry for *F* and inserts it to the AFT. It then returns to the user a pointer to the AFT copy of F's directory entry. From that point on, the user refers to the opened file, not by its name but rather through the AFT pointer supplied by the file system.
- When the user performs an operation on *F* that requires changing information in the directory entry (e.g. because the size of the file was changed), it is the AFT, not the disk, copy of F's directory entry that is modified. When the file is finally closed, the AFT entry (as possibly modified by the operations the user performed on the file while the latter was open) is written back to disk.

In this manner, the directory entry that describes a file resides in main memory (in the AFT) while a user is operating on that file.

FILE DIRECTORY STRUCTURES

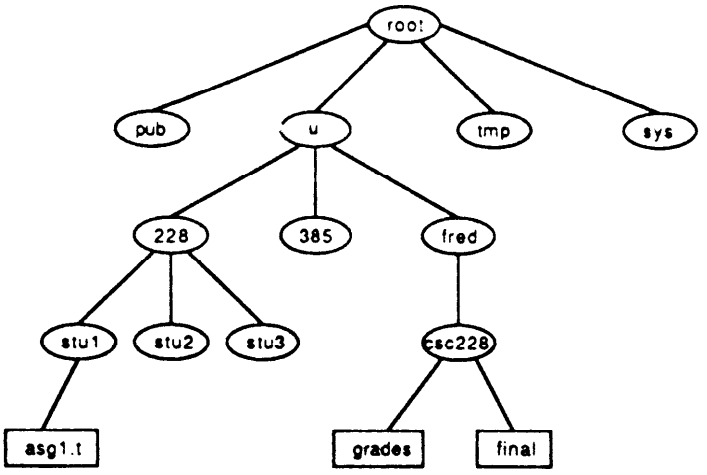
It is possible to have a single directory that contains all the files in the system. This, however, is only practical in very small, one-user environments. The main problem is that it does not allow multiple users to use the same name for different files. For example, only one file called "temp" could exist in the entire system. However, in any multi-user system, virtually all users have a file they'd like to call "temp"!

A simple solution to this problem is to give each user a separate directory. Then different users can call different files by the same name — as long as these files are in different directories. To refer to one's own files, one can use just their names. To refer to another user's file, one must specify both the user's name and the name of the file in question.

Tree-structured directories

A more general and flexible alternative is to give a hierarchical (tree) structure to the file system. When a user is registered in the system (obtains an account), s/he is assigned a directory. Within that directory the user can store files and/or create subdirectories. A subdirectory is just like a directory: its owner can put files and/or other subdirectories in it.

Thus, we can imagine the entire file system as a tree, where the leaves correspond to files in the system, while the internal (non-leaf) nodes are directories. A fragment of such a tree structured directory is illustrated in the figure below, where directories are shown as circular nodes and files as square nodes.



To name a directory or file in such a tree, one must specify the (unique) path from the root to the node that corresponds to that directory or file. Since this is inconvenient, file systems that employ this type of directory structure use the concept of "working directory". At any time a user is assigned a "working directory". The user can then specify files by giving their names relative to her/his present working directory, rather than their full path names.

Generalised directory structures

Sometimes it is useful to allow different users to share common files. For example, two programmers waking on the same programme should share the file(s) that contain that programme. For the same reason, it is useful to allow users to share directories. (In the rest of this handout, we'll use the term "file" to mean "directory or file", unless otherwise specified.)

Such sharing is impossible if the files are structured in the hierarchical way described previously. This is clear by the properties of trees: if a file is within one user's directory, it can't simultaneously be in another user's directory too.

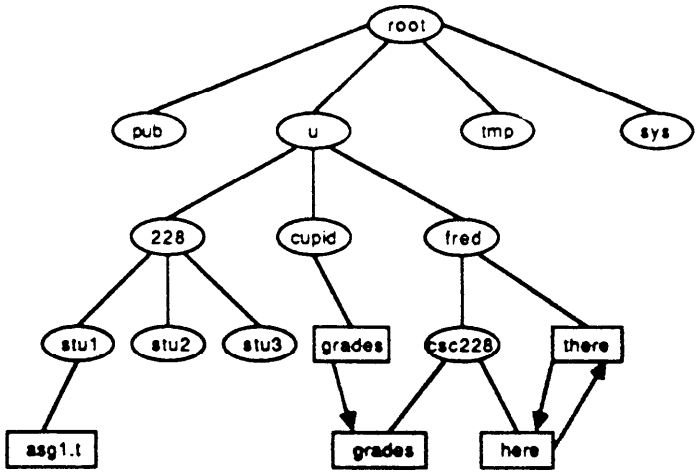
Of course, it is possible to store ***copies*** of the same file in directories of different users. But this is not the same as sharing a common file: for one thing, more storage is needed to have many copies of a single file; more significantly, if one copy of a file is changed, the modification is not carried to the other copies, while if anyone changes a shared file, all other users of the file will automatically "see" the change.

A common solution to this problem is to allow users to create ***links*** to other files. A link is a pointer to a file. There are two kinds of links:

- hard links, which specify the actual location of a file; and
- ***symbolic*** links, which specify the (full path) name of the file, using which the file system can determine the location of the file.

When a user creates a link, an entry for that link is made in the user's present working directory, as if a new file had been created.

The figure below illustrates a file system in which users can create links. A link is drawn as a broken node from which emanates a broken edge. The node contains the name of the link and the edge indicates the file to which the link is pointing.



Note that a link may be pointing to another link. This engenders the possibility of cycles in the directory structure. For instance, in the figure we have file "here" pointing to file "there" and file "there" pointing to file "here". This is a meaningless situation: what would it mean to print file "here" or "there"? As we'll see, cycles in the directory structure are a big hassle.

The fact that files can be shared by the use of links implies that there are multiple ways of referring to the same file. For instance, in the file system shown above, the path names *(root, u, fred, csc 228, grades)* and *(root, u, cupid, grades)*,† refer to the same file. *Let's* call *references* to a file all the different names for that file.

Consider a file with two references, say R_1 and R_2 and suppose a user issues the command "Delete file R_1". There are two possible interpretations for such a command:

(a) Forget about this file (and both its references).

(b) Forget about reference R_1 (but the file can be still accessed using reference R_2).

It turns out that usually the proper meaning is (b), and that's what most systems that allow for multiple references take the meaning of deletion to be. The problem is that the file system must be able to tell when the *last* reference to a file has been removed. In that event, it should return to the free pool all the space allocated to the file.

A simple solution would be to maintain, for each file, the number of references to it. This number is called the reference ***counter*** (for that file). Whenever a new reference to a file is created, the reference counter is incremented; whenever a reference to the file is deleted, the reference counter is decremented. If the reference counter should ever become zero, the space allocated to that file is returned to the free pool, as no references to the file remain.

Unfortunately this simple solution does not always work. This is because if cycles can be formed, it's possible that some files have no references to them, yet their reference counters are non-zero. For instance, in the last figure, the reference counters for files "here" and "there" are 2. If we delete references ***(root, u, fred, csc 228, here)*** and ***(root, u, fred, there),*** the two files are not accessible (there are no proper path names for them), yet their reference counters are 1, not O!

In principle, it is possible to make sure that cycles are not allowed to be formed. Each time a user tries to create a link, we can check whether the link would cause a cycle to be formed If so, an error message could be returned to the user; otherwise the link would be created. This can be done because the directory can be viewed as a directed graph (see the figure above) and there are algorithms for detecting cycles in a graph. Unfortunately, however, doing so is considerably expensive, so most file systems do not attempt to prevent cycles.

Another way of dealing with the problem of returning space to the free pool is to not bother until we have to. That is, until a user needs space for a file but the free pool does not have enough room. At that time, the file ***system starts*** a ***special*** procedure, called ***garbage collection. The*** task of this procedure is to find all the disk blocks that are not allocated to any file (that can be referenced), and return these blocks to the free pool.

† In Unix notation these path names would be: "*/u/fred/csc 228/grades*" and "*/u/cupid/grades*", respectively.

In brief outline, garbage collection works as follows. Each disk block has a special bit, used by the system for garbage collection purposes only. When the bit is zero, the block is unmarked; otherwise, it is ***marked.*** Initially, the garbage collection procedure makes all blocks unmarked. Then it examines each file in the directory and marks its blocks. When this is done, the only blocks that are still unmarked are those that are not allocated to any file. These blocks can therefore be returned to the free pool. It should be clear that garbage collection is a very time-consuming Operation.

A good practical solution combines the reference counter technique and garbage collection. Cycles in the directory structure are fairly rare and thus the reference counter technique will succeed in returning unused blocks to the free pool most of the time. Periodically (e.g. once a week, during a time that the system is not heavily used) the system runs the garbage collection procedure to collect any unused blocks that the reference counter method failed to return.

ACCESS CONTROL

Another important function of the file system is to protect files from unauthorised access.

Each user has a certain set of *access **privileges*** relative to a file. Possible access privileges are:

- The ability to read the file.
- The ability to write the file.
- The ability to append to the file (special case of previous).
- The ability to execute the file (that contains executable code).
- The ability to grant access privileges to other users for the file.
- The ability to revoke access privileges from other users for the file,

In general, we can &scribe the access privileges of users by means of a table with rows corresponding to users and columns corresponding to files. Entry (i, *j*) in the table contains the list of access rights that the user in row ***i*** has for the file ***in*** column *j*. This table is known as ***the privilege matrix*** (or ***access control matrix***). An example of such a table is shown below:

file / user	lib	grades	asgn1.t	...
fred	read write	read write	∅	
cupid	read	read append	read	
a228stu	read	∅	read write	
. . .				

The privilege matrix is quite large and most of its entries are empty. It is therefore inefficient to store it in the matrix form presented above. There are two common methods for storing the information contained in the privilege matrix:

1. Store the information with the files. In particular, for each file *F* we have a list containing (a) the users that have some access privilege(s) for *F*; and (b) for each such user, the set of privileges s/he has for F. The list is usually stored at the directory entry for *F*. When user *U* makes a request to perform an operation on file *F*, the file system checks that *U* is in the list for *F* and that *U* has the access privilege needed to perform the requested operation. If so, the system performs the operation; otherwise it returns an error message. This is the ***access fist*** approach to file security.

2. Store the privilege information with the *users. Associate* a token, called a *capability* with each (file, privilege) combination. A user U who is to have privilege P on file F is given the capability (F, P). When U wants to perform an operation on F, s/he must present the (F, P) capability to the system. The system will perform the operation only if it is presented with such a capability and P is *the privilege* needed for performing the operation; otherwise it will return *an* error message. This is the *capability* approach to file security.

The following analogy is a useful way of conceptualising the difference between these two approaches. To protect a file, we build a wall around it, and put a door in the wall. In the access list approach, we post a guard to the door. The guard has a list of the people who are allowed to enter. A person requesting entry is allowed in only if s/he is on the guard's list. In the capabilities approach, we put a lock on the door and distribute keys to those who should have access to the file. There is no need for a guard only people with keys can enter.

From this analogy it is easy to see the advantages and disadvantages of the two methods:

Entering through the *door:* is time consuming in the access list approach, because the guard must check the list; it is fast in the capability approach, because no checking of any sort is needed: the key allows direct entry.

Grunting and *revoking access:* is very easy in the access list approach: one has only to change the guard's list. It's difficult in the case of capabilities because the file system must find the users to give or take away keys.

To appreciate this last point consider the following scenario: Alice grants an access privilege to Bob; Bob further grants that privilege to Carol. Now, if Alice revokes Bob's privilege. Carol's privilege should also be removed (since she only has it be virtue of Bob's having it and Bob just lost the privilege). But Alice does not know that Carol has the privilege — she must rely on Bob's remembering that Carol has it and being kind enough to tell Alice about it Clearly, implementing such a policy requires a substantial amount of bookkeeping.

A good practical solution is obtained by combining the two approaches. More specifically, the system maintains an access list. When user U opens file F for reading or writing, the file system checks once the access list to make sure the user has the appropriate privilege. If so, it returns a capability to the user. The user can use this capability for subsequent accesses to the file, so that the file system does not have to check over and over again if the user has the privilege to access the opened file. This capability can be viewed as a temporary key: it will be good until the file is closed. At that time the lock will change and the user effectively loses the capability (until s/he opens the file again). This "key" is usually implemented as a pointer to the AFT entry of the file being opened, together with an indication whether the holder of the key (pointer) can read or write the file.

A situation one must consider in this solution is that a user's privileges to a file may be revoked while the user is still in possession of the temporary capability. For this reason, when a user's privileges to a file are revoked, all "temporary keys" given to that user for the file must be destroyed.

Non-discretionary access control policy

In the access control methods discussed so far, the owner of a file controls the granting and revoking of access privileges to that file. Such methods *are* called *discretionury.* A *non-discretionmy* method is one in which granting and revoking of access privileges to a file is controlled by the system, not any user. Such methods are used in environments with high standards of security — military establishments, government agencies, certain aspects of business etc.

Following is a description of the non-discretionary access control scheme used by the US Department of Defense. Each file is assigned a security *classification* defined as a pair (L, C) where L is called the *level* and C the *category* of the security classification. The level has one of the following values, ordered as indicated:

unclassified < confidential < secret < top secret

The category is a set of user groups. Each user is also assigned a security classification (or "clearance") — i.e. a *(level, category) pair.*

If $(L,$ C) and (L', C') are security classifications we write $(L, C) \geq (L', C')$ (read: (L, C) is at least as strong as (L', C')) *if* $L \geq L'$ and C *supset* C' (recall that C, C' are sets of user groups).

The access control method consists of two rules:

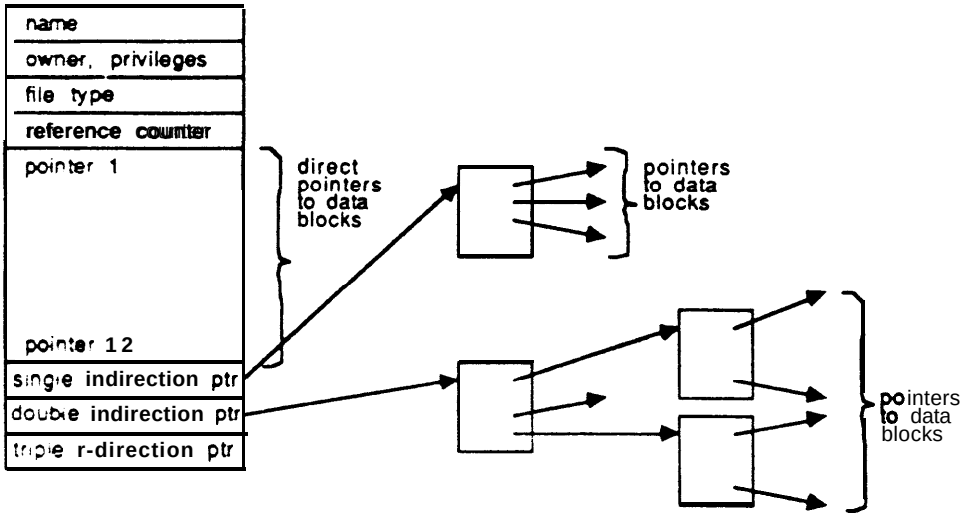
- ***The read up rule:*** *A* user with security classification (L_u, C_u) can read a file with security classification (L_f, C_f) if and only if $(L_u, C_u) \geq (L_f, C_f)$

- The *write down* rule: A user who has access to a file can't copy the file into another with weaker classification. (So a user with access to a file can't compromise the security of the file indirectly, by making a copy of the file available to other users with weaker security clearances.)

SOME ASPECTS OF THE UNIX FILE SYSTEM

In this section we concentrate on the file system of the Unix operating system. In no small measure, the success enjoyed by Unix, is owed to its very well designed file system.

In Unix, disk space is allocated using the indexed method. Each file is described by an index block, called *inode* in Unix parlance. The (approximate) information stored at an inode is illustrated below.



In addition to information such as the name, owner, type and reference counter, the inode contains 15 pointers to other blocks. The first 12 pointers are direct pointers to (the first 12) data blocks of the file. Pointer 13 is a single indirection pointer; i.e. it points to a block that contains pointers to the file's subsequent data blocks. Pointer 14 is a double indirection pointer; i.e. it points to a block that contains single indirection pointers. Finally, pointer 15 is a triple indirection pointer; i.e. points to a block that contains double indirection pointers.

In the 4.2BSD implementation of Unix (the one that our computers are running), the block size is 4Kb (i.e. 2^{12} bytes); a pointer to a block is 32 bits (4 bytes). Thus a block can hold 2^{10} block addresses. Therefore, in this implementation

12×2^{12} bytes are accessible directly.
$2^{10} \times 2^{12}$ bytes are accessible by single indirection.
$2^{10} \times 2^{10} \times 2^{12}$ bytes are accessible by double indirection.
$2^{10} \times 2^{10} \times 2^{10} \times 2^{12}$ bytes are accessible by triple indirection.

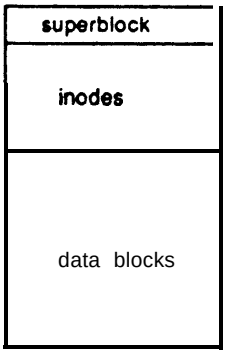
The total number of bytes accessible directly, by single and double indirection exceeds 2^{32} (4Gb!). This is the limit on the size of files allowed in Unix.† Thus, the triple indirection pointer is never used.

With this method, blocks of short files (with up to 12×2^{12}, i.e. 49,152 bytes) can be found directly from information in the inode; for larger files one or two additional block accesses are required to locate a block of the file.

† The reason for this limit is that Unix uses a 32-bit word to store the value of the cursor (present byte offset) for an opened file. Thus, no more than 2^{32} bytes can be addressed.

Layout of blocks on disk

As always we think of the disk as a large linear array of blocks. The overall layout of blocks on a disk in the Unix system is shown below.



(NB: The indirect blocks to which pointers 13.14 and 15 in an inode may point are *data* blocks, not inodes.)

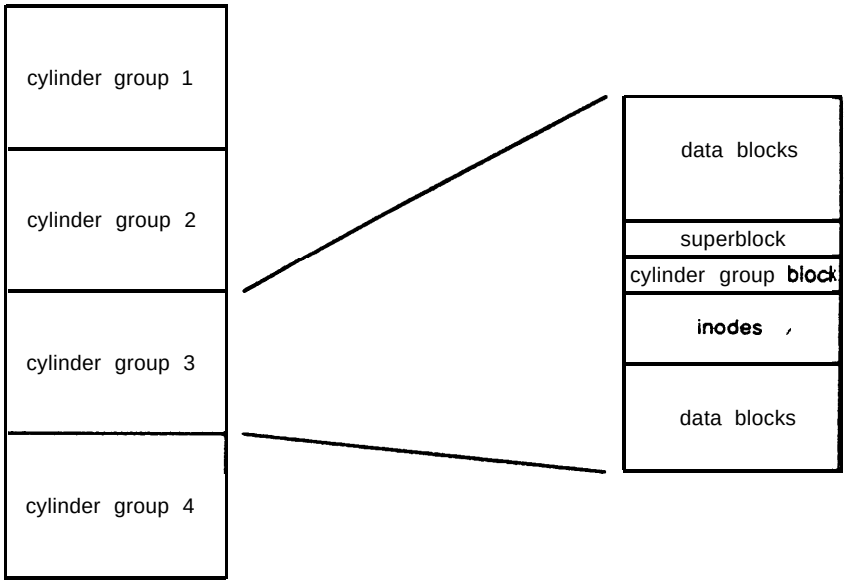
The superblock contains general information about the blocks on this disk. In particular, it contains the free pool (in the form of a bitmap, or in the form of a list).

This is the layout used in the implementations of Unix Versions 6 and 7. It has two problems:

1) Reliability: System crashes and operating system bugs occasionally cause the superblock to be destroyed. In that event, we lose all the information about which disk blocks are free and which aren't The free pool must be reconstructed by a time-consuming examination of all disk blocks.

2) Efficiency: Data blocks of any given file might be scattered all over the disk, so in processing a file much time may be spent on seeks.

The 4.2BSD implementation of Unix uses a more sophisticated space allocation method to alleviate these problems. Each disk is partitioned into a number of *cylinder groups.* A cylinder group consists of all the blocks in a number of consecutive cylinders. The overall layout of cylinder groups within a disk and the structure of a cylinder group is shown in the figure below.



Each cylinder group contains a copy of the disk's superblock. Thus there are several copies of that block. This redundancy is introduced to enhance reliability: if one copy is destroyed we can use another to reconstruct it.† Also,

† In fact, the superblock copies are kept on different relative positions in different cylinder groups, to minimise the chance of a disk head crash

each cylinder group has its own cylinder group block, describing the state of the blocks (free or used) in that group.

In allocating blocks to a file, the system tries to allocate as many as possible from within a cylinder group. When no more blocks are available, it continues the allocation at another cylinder group (choosing one that's as empty as possible — so as to reduce the chance of having to allocate blocks in yet another cylinder group for the same file — and as close as possible to the present cylinder group — so as to reduce the seek time in moving from one to the otherj.

In this manner, the blocks of a file reside in relatively few different cylinder groups that tend to be close together. This reduces the amount of time spent on seeks.

This implementation has dramatically improved the performance of the Unix file system.

The Unix directory structure and its implementation

Basically the Unix file directory is structured as a tree. However, links can also be created, and thus shared files/directories can be created and even cycles in the directory structure introduced

There is no single file that contains the entire file directory structure. Rather, each node in the tree (directories included) is implemented as an ordinary file. The inode of a file that's actually a directory has an indication to that effect in the "type" field.

The file that represents directory ***D***, contains one entry for each file, directory and link in ***D***. In addition, it contains two special entries, denoted "." and "..". The former is an entry for the directory ***D*** itself (self-reference) and the latter is an entry for ***D's*** parent in the tree structure.

Links

Unix allows the creation of both hard and symbolic links. Hard links can be created for files only; symbolic links can be created for either files or directories. As was said before, the creation of (symbolic) links can introduce cycles to the directory structure. Unix deals with this problem in a rather crude way: if the file system tries to find an actual file or directory following a path that contains more than a certain number of symbolic links (eight), it gives up and returns an error message! This is, of course, crude, but is better than winding up in an infinite loop following directory cycles.

Access privileges in Unix

Unix uses the access list method. One problem with that method is that if many users have access privileges on some particular file, the access list for that file will be quite long. This is inconvenient since the access list must be stored at an inode (and be brought to the AFT in main memory, when the file is opened).

In Unix this is solved as follows: Each file has a unique owner (typically the user who created it). The set of users (currently registered in the system) is partitioned into groups. Each user belongs to one and only one group. (All accounts for a course, for instance, are in the same group.)

As far as the protection of a particular file goes, the population of users is divided to three (disjoint) categories:

1. The owner of the file.
2. All other users in the Same group as the owner (but excluding the owner).
3. Everybody else (all users in different groups than the owner).

There are three access privileges in Unix read, write and execute (x). For a file ***F***, the owner can specify which privilege-s each of the previous 3 categories has. Thus the access list for a file consists of nine bits:

r w x r w x r w x

damaging all copies of the superblock.

Only the owner of a file can change these bits. (Thus only the owner can grant and revoke access privileges.)

In this way the access list is kept short. This not only solves the problem of space, but also makes it easy to check whether a user has access to a file.

References

Quarterman, J.S., Silberschatz, A., and Peterson, J.L., "4.2BSD and 4.3BSD as Examples of the UNIX System," *ACM Computing Surveys* 17(4), 1985, pp. 379-418.