Chapter 6

File Systems

Support for files is an abstraction provided by the operating system, and does not entail any real resource management. If disk space is available, it is used, else the service cannot be provided. There is no room for manipulation as in scheduling (by timesharing) or in memory management (by paging and swapping). However there is some scope for various ways to organize the data when implementing the file abstraction.

6.1 What is a File?

The most important characteristics are being named and persistent

A file is a named persistent sequential (structured) data repository.

The attributes of being named and being persistent go hand in hand. The idea is that files can be used to store data for long periods of time, and specifically, for longer than the runtimes of the processes that create them. Thus one process can create a file, and another process may re-access the file using its name.

The attribute of being sequential means that data within the file can be identified by its offset from the beginning of the file.

As for structure, at least one structure is typically required by the operating system: that of an executable file. The support for other structures is optional. Examples include:

- Unix: all files are a sequence of bytes, with no structure as far as the operating system is concerned. The only operations are to read and write bytes. Interpretation of the data is left to the application using it. (And when the file is an executable, the application that needs to interpret the structure happens to be the operating system).
- IBM mainframes: the operating system supports lots of options, including fixed

or variable-size records, indexes, etc. Thus support for higher-level operations is possible, including "read the next record" or "read the record with key X".

- Macintosh OS: executables have two "forks": one containing code, the other labels used in the user interface. Users can change labels appearing on buttons in the application's graphical user interface (e.g. to support different languages) without access to the source code.
- Windows NTFS: files are considered as a set of attribute/value pairs. In particular, each file has an "unnamed data attribute" that contains its contents. But it can also have multiple additional named data attributes, e.g. to store the file's icon or some other file-specific information. Files also have a host of system-defined attributes.

The extreme in terms of structure is a full fledged database. As the organization and use of databases is quite different from that encountered in general-purpose systems, it is common to use a special database management system (DBMS) in lieu of the operating system. We shall not discuss database technology here.

The most important attributes are permissions and data layout

Given that files are abstract objects, one can ask what attributes are kept about them. While there are differences among systems, the main ones are

`Owner:` the user who owns this file.

`Permissions:` who is allowed to access this file.

`Modification time:` when this file was last modified.

`Size:` how many bytes of data are there.

`Data location:` where on the disk the file's data is stored.

As this is data *about* the file maintained by the operating system, rather than user data that is stored *within* the file, it is sometimes referred to as the file's *metadata*.

Exercise 122 *What are these data items useful for?*

Exercise 123 *And how about the file's name? Why is it not here?*

Exercise 124 *The Unix `stat` system call provides information about files (see man `stat`). Why is the location of the data on the disk not provided?*

The most important operations are reading and writing

Given that files are abstract objects, one can also ask what operations are supported on these objects. In a nutshell, the main ones are

`Open`: gain access to a file.

`Close`: relinquish access to a file.

`Read`: read data from a file, usually from the current position.

`Write`: write data to a file, usually at the current position.

`Append`: add data at the end of a file.

`Seek`: move to a specific position in a file.

`Rewind`: return to the beginning of the file.

`Set attributes`: e.g. to change the access permissions.

`Rename`: change the name of the file.

Exercise 125 *Is this set of operations minimal, or can some of them be implemented using others?*

6.2 File Naming

As noted above, an important characteristic of files is that they have names. These are typically organized in directories. But internally, files (and directories) are represented by a data structure containing the attributes listed above. Naming is actually a mapping from names — that is, strings given by human users — to these internal representations. In Unix, this data structure is called an inode, and we'll use this name in what follows as shorthand for "a file or directory's internal representation".

6.2.1 Directories

Directories provide a hierarchical structure

The name space for files can be *flat*, meaning that all files are listed in one long list. This has two disadvantages:

- There can be only one file with a given name in the system. For example, it is not possible for different users to have distinct files named "ex1.c".
- The list can be very long and unorganized.

The alternative is to use a hierarchical structure of directories. This structure reflects organization and logical relations: for example, each user will have his own private directory. In addition, files with the same name may appear in different directories.

Using directories also creates an opportunity for supporting collective operations on sets of related files: for example, it is possible to delete all the files in a given directory.

With a hierarchical structure files are identified by the path from the root, through several levels of directories, to the file. Each directory contains a list of those files and subdirectories that are contained in it. Technically, the directory maps the names of these files and subdirectories to the internal entities known to the file systems — that is, to their inodes. In fact, directories are just like files, except that the data stored in them is not user data but rather file system data, namely this mapping. The hierarchy is created by having names that refer to subdirectories.

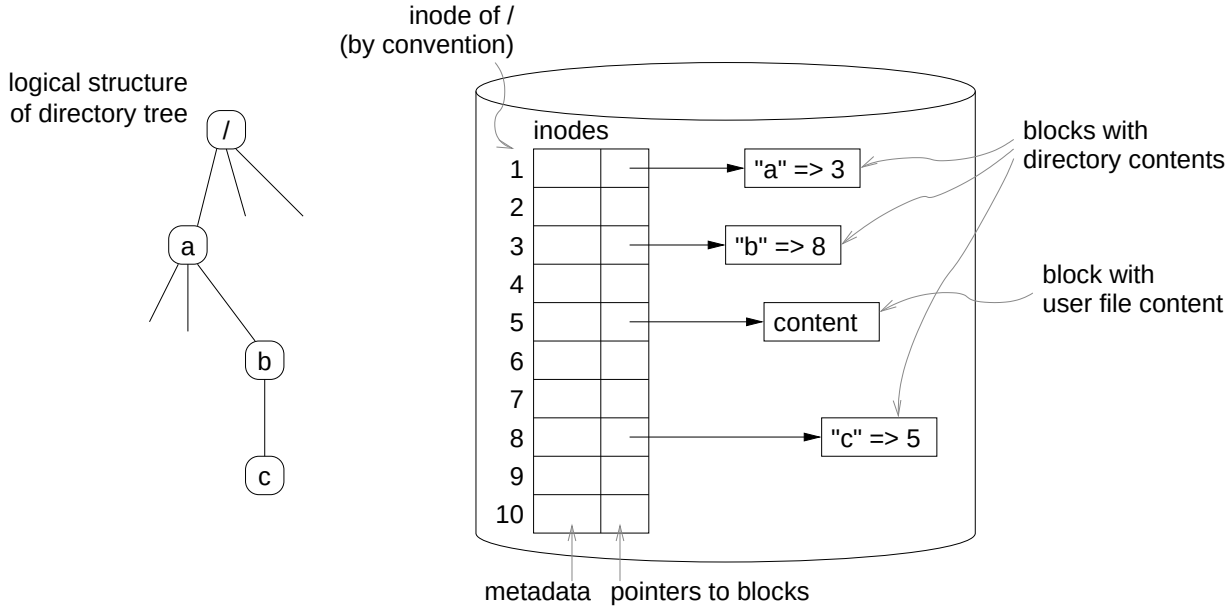
Exercise 126 *What happens if we create a cycle of directories? What is a simple way to prevent this?*

Names are mapped to inodes recursively

The system identifies files by their full *path*, that is, the file's name concatenated to the list of directories traversed to get to it. This always starts from a distinguished *root directory*.

Exercise 127 *So how does the system find the root directory itself?*

Assume a full path `/a/b/c` is given. To find this file, we need to perform the following:



1. Read the inode of the root / (assume it is the first one in the list of inodes), and use it to find the disk blocks storing its contents, i.e. the list of subdirectories and files in it.

2. Read these blocks and search for the entry `a`. This entry will map `/a` to its inode. Assume this is inode number 3.
3. Read inode 3 (which represents `/a`), and use it to find its blocks.
4. Read the blocks and search for subdirectory `b`. Assume it is mapped to inode 8.
5. Read inode 8, which we now know to represent `/a/b`.
6. Read the blocks of `/a/b`, and search for entry `c`. This will provide the inode of `/a/b/c` that we are looking for, which contains the list of blocks that hold this file's contents.
7. To actually gain access to `/a/b/c`, we need to read its inode and verify that the access permissions are appropriate. In fact, this should be done in each of the steps involved with reading inodes, to verify that the user is allowed to see this data. This is discussed further in Chapter 7.

Note that there are 2 disk accesses per element in the path: one for the inode, and the other for the contents.

A possible shortcut is to start from the current directory (also called the working directory) rather than from the root. This obviously requires the inode of the current directory to be available. In Unix, the default current directory when a user logs onto the system is that user's home directory.

Exercise 128 *Are there situations in which the number of disk accesses when parsing a file name is different from two per path element?*

Exercise 129 *The contents of a directory is the mapping from names (strings of characters) to inode (e.g. integers interpreted as an index into a table). How would you implement this? Recall that most file names are short, but you need to handle arbitrarily long names efficiently. Also, you need to handle dynamic insertions and deletions from the directory.*

Exercise 130 *In unix, the contents of a directory is simply a list of mappings from name to inode. An alternative is to use a hash table. What are the advantages and disadvantages of such a design?*

Note that the process described above may also fail. For example, the requested name may not be listed in the directory. Alternatively, the name may be there but the user may not have the required permission to access the file. If something like this happens, the system call that is trying to gain access to the file will fail. It is up to the application that called this system call to print en error message or do something else.

6.2.2 Links

Files can have more than one name

The mapping from a user-defined name string to an inode) is called a link. In principle, it is possible to have multiple strings mapped to the same inode. This causes the file to have multiple names. And if the names appear in different directories, it will have multiple different paths.

Exercise 131 *Why in the world would you want a file to have multiple names?*

Special care must be taken when deleting (unlinking) a file. If it has multiple links, and one is being removed, we should not delete the file's contents — as they are still accessible using the other links. The inode therefore has a counter of how many links it has, and the data itself is only removed when this count hits zero.

Exercise 132 *An important operation on files is renaming them. Is this really an operation on the file? How it is implemented?*

Soft links are flexible but problematic

The links described above, implemented as mappings in a directory, are called hard links. An alternative is soft links: a directory entry that is actually an indirection, not a real link (in Windows systems, this is called a "shortcut"). This means that instead of mapping the name to an inode, the name is mapped to an alternative path. When a soft link is encountered in the process of parsing a path name, the current path is replaced by the one indicated in the link.

Exercise 133 *What are advantages and dangers of soft links?*

6.2.3 Alternatives for File Identification

The reason for giving files names is to make them findable. But when you have lots of files, accumulated over many years, you might forget the name you chose. And names are typically constrained to be quite short (if not by the system, then by your desire not to type too much).

It might be better to identify files by association

An alternative to names is to identify files by association. Specifically, you will probably think of files in the context of what you were working on at the time. So an association with this context will make the desired files easy to find.

One simple interface suggested to accomplish this is "lifestreams". With this, files are shown in sequence according to when they were used, with the newest ones at the

front. The user can use the mouse to move backwards or forward in time and view different files [5].

A more sophisticated interface is the calendarial file access mechanism developed at Ricoh Research Center [7]. The interface is a calendar, with a box for each day, organized into weeks, months, and years. Each box contains information about scheduled activities in that day, e.g. meetings and deadlines. It also contains thumbnail images of the first pages of document files accessed on that day. Thus when looking for a file, you can find it by searching for the meeting in which it was discussed. Importantly, and based on the fact that Ricoh is a manufacturer of photocopiers and other office equipment, this includes all documents you worked with, not only those you viewed or edited on your computer. In a 3-year usage trial, 38% of accesses using the system were to documents only one week old, showing that in many cases users preferred this interface to the conventional one even for relatively fresh documents.

Or to just search

Another recent alternative is to use keyword search. This is based on the success of web search engines, that index billions of web pages, and provide a list of relevant pages when queried. In principle, the same can be done for files in a file system. The problem is how to rank the files and show the most relevant on top.

To read more: The opposition to using file names, and preferring search procedures, is promoted by Raskin, the creator of the Mac interface [11].

6.3 Access to File Data

The main function of a file system is to store data in files. But files are an abstraction which needs to be mapped to and implemented with the available hardware. This involves the allocation of disk blocks to files, or, viewed the other way, the mapping of files into the disk. It also involves the actual read and write operations, and how to optimize them. One important optimization is to avoid disk access altogether by caching data in memory.

As in other areas, there are many options and alternatives. But in the area of file systems, there is more data about actual usage patterns than in other areas. Such data is important for the design and evaluation of file systems, and using it ensures that the selected policies do indeed provide good performance.

> **Background: direct memory access (DMA)**
> Getting the data on or off the disk is only one side of the I/O operation. The other side is accessing the appropriate memory buffer. The question of how this is done depends on the hardware, and has a great impact on system performance.
>
> If only the processor can access memory the only option is *programmed I/O*. This means that the CPU (running operating system code) accepts each word of data as it comes off

the disk, and stores it at the required address. This has the obvious disadvantage of keeping the CPU busy throughout the duration of the I/O operation, so no overlap with other computation is possible.

In modern systems components who need it typically have Direct Memory Access (DMA), so they do not have to go through the CPU (this is true for both disks and network interfaces). Thus when the operating system wants to perform an I/O operation, it only has to activate the disk controller, and tell it what to do. The operating system then blocks the process that requested this I/O operation, and frees the CPU to run another ready process. In the meantime, the disk controller positions the heads, accesses the disk, and transfers the data between the disk and the memory. When the transfer is complete, the disk controller interrupts the CPU. The operating system interrupt handler unblocks the waiting process, and puts it on the ready queue.

6.3.1 Data Access

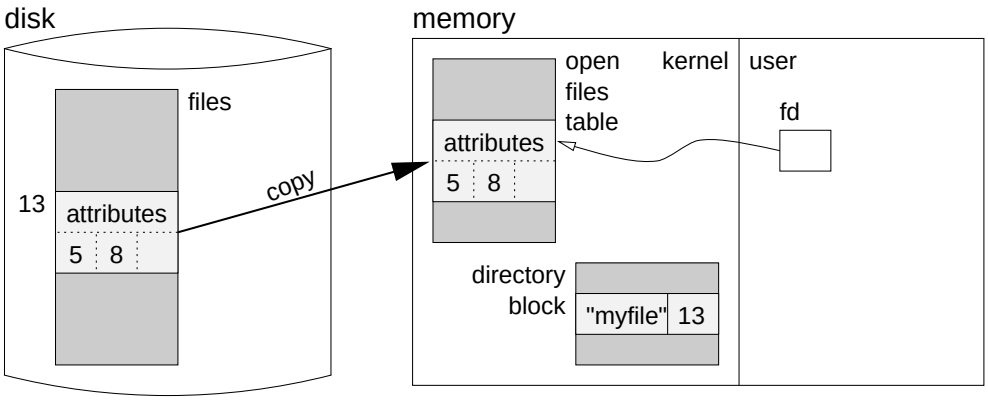
To use files, users use system calls that correspond to the operations listed in Section 6.1. This section explains how the operating system implements these operations.

Opening a file sets things up for future access

Most systems require files to be *opened* before they can be accessed. For example, using Unix-like notation, the process may perform the system call

```
fd=open("myfile",R)
```

This leads to the following sequence of actions:



1. The file system reads the current directory, and finds that “myfile” is represented internally by entry 13 in the list of files maintained on the disk.
2. Entry 13 from that data structure is read from the disk and copied into the kernel’s open files table. This includes various file attributes, such as the file’s owner and its access permissions, and a list of the file blocks.

3. The user's access rights are checked against the file's permissions to ascertain that reading (`R`) is allowed.
4. The user's variable `fd` (for "file descriptor") is made to point to the allocated entry in the open files table. This serves as a handle to tell the system what file the user is trying to access in subsequent `read` or `write` system calls, and to prove that permission to access this file has been obtained, but without letting user code obtain actual access to kernel data.

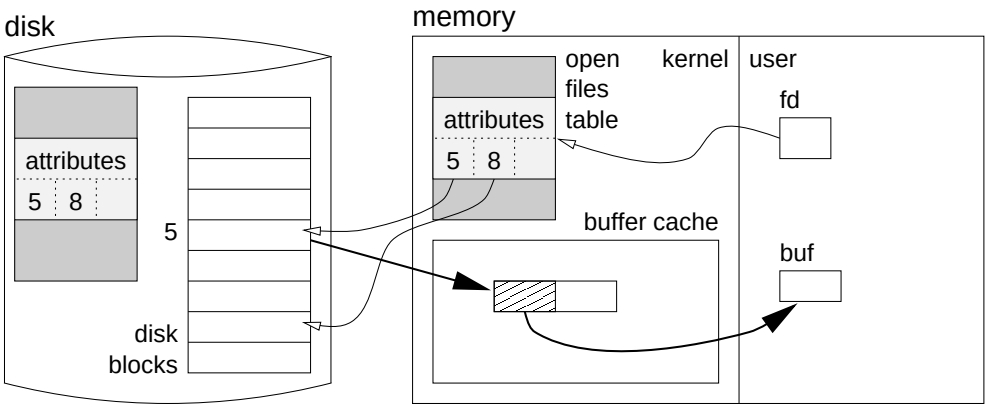
The reason for opening files is that the above operations may be quite time consuming, as they may involve a number of disk accesses to map the file name to its inode and to obtain the file's attributes. Thus it is desirable to perform them once at the outset, rather then doing them again and again for each access. `open` returns a handle to the open file, in the form of the file descriptor, which can then be used to access the file many times.

Access to disk blocks uses the buffer cache

Now the process performs the system call

```
read(fd,buf,100)
```

which means that 100 bytes should be read from the file indicated by `fd` into the memory buffer `buf`.



The argument `fd` identifies the open file by pointing into the kernel's open files table. Using it, the system gains access to the list of blocks that contain the file's data. In our example, it turns out that the first data block is disk block number 5. The file system therefore reads disk block number 5 into its *buffer cache*. This is a region of memory where disk blocks are cached. As this takes a long time (on the order of milliseconds), the operating system typically blocks the operation waiting for it to complete, and does something else in the meanwhile, like scheduling another process to run.

When the disk interrupts the CPU to signal that the data transfer has completed, handling of the `read` system call resumes. In particular, the desired range of 100 bytes is copied into the user's memory at the address indicated by `buf`. If additional bytes from this block will be requested later, the block will be found in the buffer cache, saving the overhead of an additional disk access.

Exercise 134 *Is it possible to read the desired block directly into the user's buffer, and save the overhead of copying?*

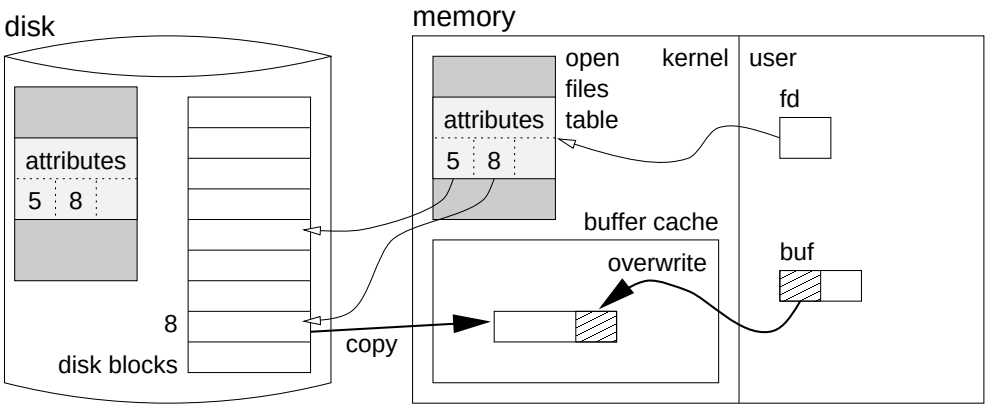
Writing may require new blocks to be allocated

Now suppose the process wants to write a few bytes. Let's assume we want to write 100 bytes, starting with byte 2000 in the file. This will be expressed by the pair of system calls

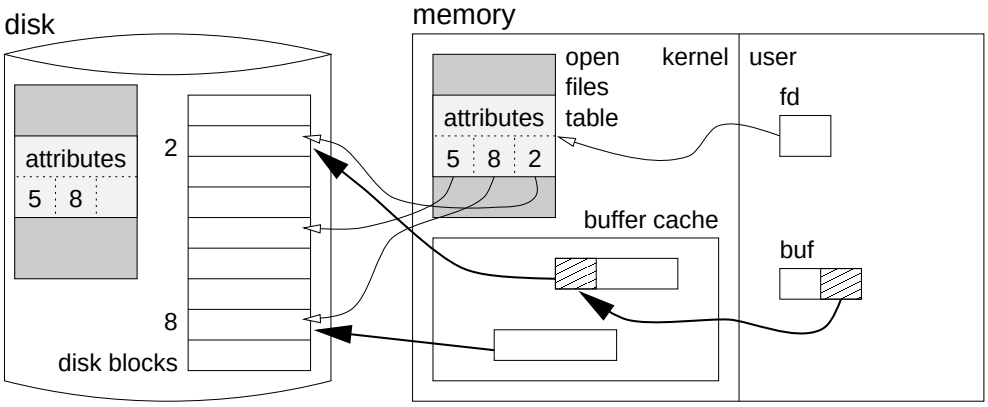
```
seek(fd,2000)
write(fd,buf,100)
```

Let's also assume that each disk block is 1024 bytes. Therefore the data we want to write spans the end of the second block to the beginning of the third block.

The problem is that disk accesses are done in fixed blocks, and we only want to write part of such a block. Therefore the full block must first be read into the buffer cache. Then the part being written is modified by overwriting it with the new data. In our example, this is done with the second block of the file, which happens to be block 8 on the disk.



The rest of the data should go into the third block, but the file currently only has two blocks. Therefore a third block must be allocated from the pool of free blocks. Let's assume that block number 2 on the disk was free, and that this block was allocated. As this is a new block, there is no need to read it from the disk before modifying it — we just allocate a block in the buffer cache, prescribe that it now represents block number 2, and copy the requested data to it. Finally, the modified blocks are written back to the disk.



Note that the copy of the file's inode was also modified, to reflect the allocation of a new block. Therefore this too must be copied back to the disk. Likewise, the data structure used to keep track of free blocks needs to be updated on the disk as well.

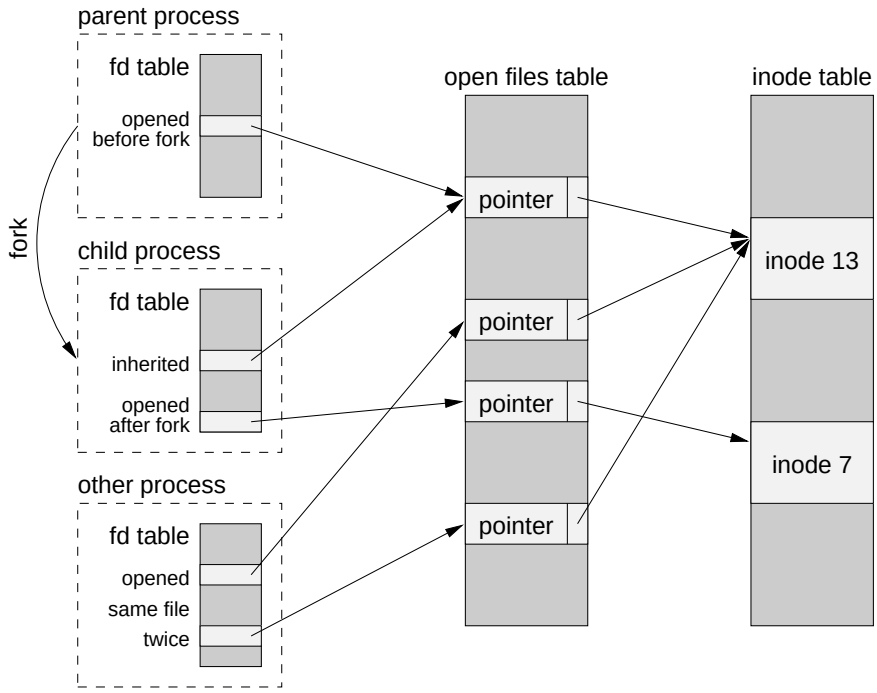
The location in the file is maintained by the system

You might have noticed that the `read` system call provides a buffer address for placing the data in the user's memory, but does not indicate the offset in the file from which the data should be taken. This reflects common usage where files are accessed sequentially, and each access simply continues where the previous one ended. The operating system maintains the current offset into the file (sometimes called the *file pointer*), and updates it at the end of each operation.

If random access is required, the process can set the file pointer to any desired value by using the `seek` system call (random here means arbitrary, not indeterminate!).

Exercise 135 *What happens (or should happen) if you seek beyond the current end of the file, and then write some data?*

> **Example: Unix allows the file pointer to be shared**
>
> An interesting option is to share a file pointer among multiple processes. For example, this is useful in writing a log file that is shared by several processes. If each process has its own file pointer, there is a danger that one process will overwrite log entries written by another process. But if the file pointer is shared, each new log entry will indeed be added to the end of the file.
>
> To implement this, Unix uses a set of three tables to control access to files. The first is the in-core inode table, which contains the inodes of open files (recall that an inode is the internal structure used to represent files, and contains the file's metadata as outlined in Section 6.1). Each file may appear at most once in this table.



The second table is the open files table. An entry in this table is allocated every time a file is opened. These entries contain three main pieces of data:

- An indication of whether the file was opened for reading or for writing
- An offset (sometimes called the "file pointer") storing the current position within the file.
- A pointer to the file's inode.

A file may be opened multiple times by the same process or by different processes, so there can be multiple open file entries pointing to the same inode.

The third table is the file descriptors table. There is a separate file descriptor table for each process in the system. When a file is opened, the system finds an unused slot in the opening process's file descriptor table, and uses it to store a pointer to the new entry it creates in the open files table. The index of this slot is the return value of the `open` system call, and serves as a handle to get to the file. The first three indices are by convention pre-allocated to standard input, standard output, and standard error.

The important thing about the file descriptors table is that it is inherited across forks. Thus if a process opens a file and then forks, the child process will also have a file descriptor pointing to the same entry in the open files table. The two processes can then share the same file pointer by using these file descriptors. If either process opens a file *after* the fork, the associated file pointer is not shared.

Exercise 136 *Given that multiple file descriptors can point to the same open file entry, and multiple open file entries can point to the same inode, how are entries freed? Specifically,*

when a process closes a file descriptor, how can the system know whether it should free the open file entry and/or the inode?

Exercise 137 *Assuming we do not care about the option of sharing the file pointer, could* open *just return the index into the open files table?*

6.3.2 Caching and Prefetching

Disk I/O is substantially slower than processing, and the gap is growing: CPUs are becoming faster much faster than disks. This is why it makes sense to perform a context switch when waiting for I/O. It also means that some effort should be invested in making I/O faster.

Caching is instrumental in reducing disk accesses

As mentioned above, operating systems typically place a buffer cache between the disk and the user. All file operations pass through the buffer cache. The use of a buffer cache is required because disk access is performed in predefined blocks, that do not necessarily match the application's requests. However, there are a few additional important benefits:

- If an application requests only a small part of a block, the whole block has to be read anyway. By caching it (rather than throwing it away after satisfying the request) the operating system can later serve additional requests from the same block without any additional disk accesses. In this way small requests are aggregated and the disk access overhead is amortized rather than being duplicated.
- In some cases several processes may access the same disk block; examples include loading an executable file or reading from a database. If the blocks are cached when the first process reads them off the disk, subsequent accesses will hit them in the cache and not need to re-access the disk.
- A lot of data that is written to files is actually transient, and need not be kept for long periods of time. For example, an application may store some data in a temporary file, and then read it back and delete the file. If the file's blocks are initially stored in the buffer cache, rather than being written to the disk immediately, it is possible that the file will be deleted while they are still there. In that case, there is no need to write them to the disk at all. The same holds for data that is overwritten after a short time.

Data about file system usage in working Unix 4.2 BSD systems was collected and analyzed by Ousterhout and his students [9, 2]. They found that a suitably-sized buffer cache can eliminate 65–90% of the disk accesses. This is attributed to the

reasons listed above. Specifically, the analysis showed that 20–30% of newly-written information is deleted or overwritten within 30 seconds, and 50% is deleted or overwritten within 5 minutes. In addition, about 2/3 of all the data transferred was in sequential access of files. In files that were opened for reading or writing, but not both, 91–98% of the accesses were sequential. In files that were opened for both reading and writing, this dropped to 19–35%.

The downside of caching disk blocks is that the system becomes more vulnerable to data loss in case of a system crash. Therefore it is necessary to periodically flush all modified disk blocks to the disk, thus reducing the risk. This operation is called *disk synchronization*.

Exercise 138 *It may happen that a block has to be read, but there is no space for it in the buffer cache, and another block has to be evicted (as in paging memory). Which block should be chosen? Hint: LRU is often used. Why is this possible for files, but not for paging?*

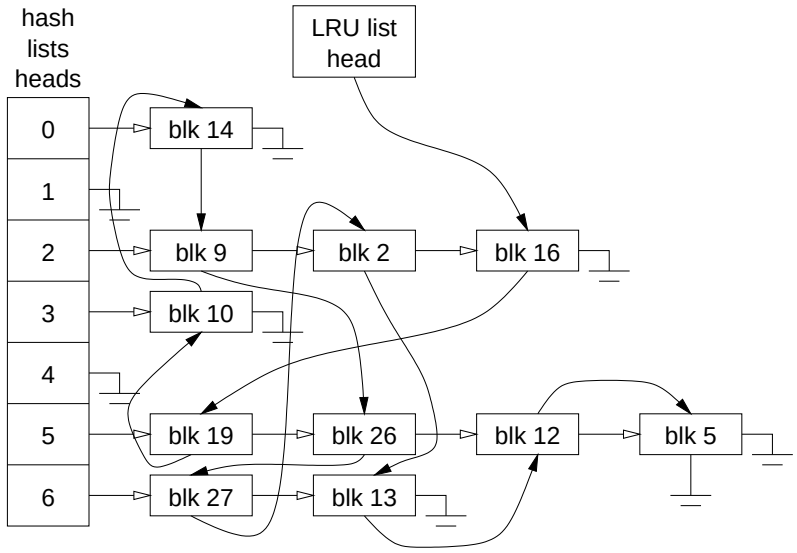
Example: the Unix Buffer Cache Data Structure

A cache, by definition, is associative: blocks are stored randomly (at least in a fully associative cache), but need to be accessed by their address. In a hardware cache the search must be performed in hardware, so costs dictate a sharp reduction in associativity (leading, in turn, to more conflicts and reduced utilization). But the buffer cache is maintained by the operating system, so this is not needed.

The data structure used by Unix[1] to implement the buffer cache is somewhat involved, because two separate access scenarios need to be supported. First, data blocks need to be accessed according to their disk address. This associative mode is implemented by hashing the disk address, and linking the data blocks according to the hash key. Second, blocks need to be listed according to the time of their last access in order to implement the LRU replacement algorithm. This is implemented by keeping all blocks on a global LRU list. Thus each block is actually part of two linked lists: one based on its hashed address, and the other based on the global access order.

For example, the following simplified picture shows a possible linking of 11 data blocks, where the hashing is done by taking the block number modulo 7.

[1]Strictly speaking, this description relates to older versions of the system. Modern Unix variants typically combine the buffer cache with the virtual memory mechanisms.



Prefetching can overlap I/O with computation

The fact that most of the data transferred is in sequential access implies locality. This suggests that if the beginning of a block is accessed, the rest will also be accessed, and therefore the block should be cached. But it also suggests that the operating system can guess that the *next* block will also be accessed. The operating system can therefore prepare the next block *even before is is requested*. This is called prefetching.

Prefetching does not reduce the number of disk accesses. In fact, it runs the risk of increasing the number of disk accesses, for two reasons: first, it may happen that the guess was wrong and the process does not make any accesses to the prefetched block, and second, reading the prefetched block into the buffer cache may displace another block that will be accesses in the future. However, it does have the potential to significantly reduce the time of I/O operations as observed by the process. This is due to the fact that the I/O was started ahead of time, and may even complete by the time it is requested. Thus prefetching overlaps I/O with the computation of the same process. It is similar to asynchronous I/O, but does not require any specific coding by the programmer.

Exercise 139 *Asynchronous I/O allows a process to continue with its computation, rather than being blocked while waiting for the I/O to complete. What programming interfaces are needed to support this?*

6.3.3 Memory-Mapped Files

An alternative to the whole mechanism described above, which is used by many modern systems, is to map files to memory. This relies on the analogy between handling file access and handling virtual memory with paging.

File layout is similar to virtual memory layout

As described in the previous pages, implementing the file abstraction has the following attributes:

- Files are continuous sequences of data.
- They are broken into fixed-size blocks.
- These blocks are mapped to arbitrary locations in the disk, and the mapping is stored in a table.

But this corresponds directly to virtual memory, where continuous logical memory segments are broken into pages and mapped to memory frames using a page table. Instead of implementing duplicated mechanisms, it is better to use one to implement the other.

Mapping files to memory uses paging to implement I/O

The idea is simple: when a file is opened, create a memory segment and define the file as the disk backup for this segment. In principle, this involves little more than the allocation of a page table, and initializing all the entries to invalid (that is, the data is on disk and not in memory).

A `read` or `write` system call is then reduced to just copying the data from or to this mapped memory segment. If the data is not already there, this will cause a page fault. The page fault handler will use the inode to find the actual data, and transfer it from the disk. The system call will then be able to proceed with copying it.

Memory mapped files are more efficient

An important benefit of using memory-mapped files is that this avoids the need to set aside some of the computer's physical memory for the buffer cache. Instead, the buffered disk blocks reside in the address spaces of the processes that use them. The portion of the physical memory devoted to such blocks can change dynamically according to usage, by virtue of the paging mechanism.

Moreover, if memory mapping is exposed as part of the interface, copying the data may be avoided. For example, if a process maps a file instead of reading it piecemeal, the process is given a pointer that indicates where the file data was mapped. Thus it can access the data directly using this pointer, without first copying it into a separate buffer as would be done by a `read` system call.

Exercise 140 *What happens if two distinct processes map the same file?*

To read more: See the man page for `mmap`.

6.4 Storing Files on Disk

The medium of choice for persistent storage is magnetic disks. This may change. In the past, it was tapes. In the future, it may be flash memory and optical disks such as DVDs. Each medium has its constraints and requires different optimizations. The discussion here is geared towards disks.

6.4.1 Mapping File Blocks

OK, so we know about accessing disk blocks. But how do we find the blocks that together constitute the file? And how do we find the right one if we want to access the file at a particular offset?

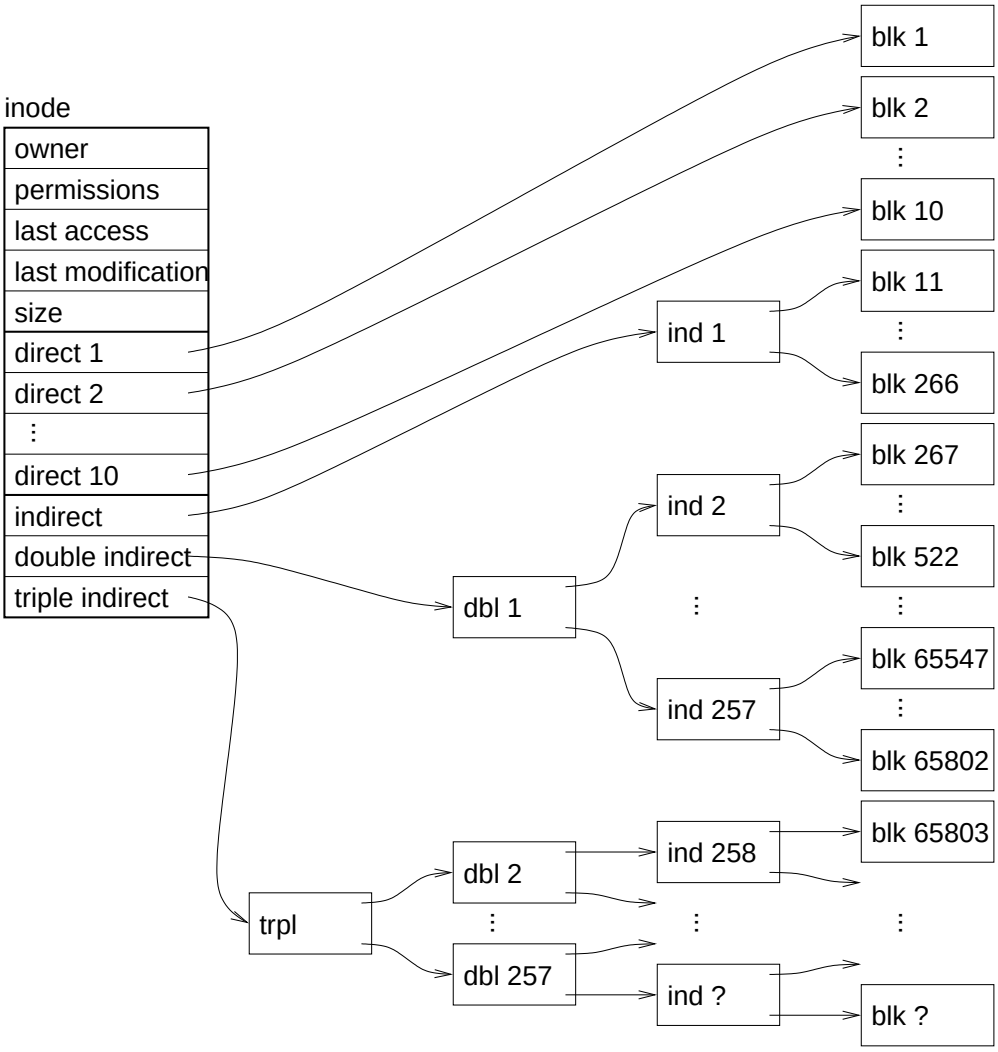
The Unix inode contains a hierarchical index

In Unix files are represented internally by a structure known as an inode. Inode stands for "index node", because apart from other file metadata, it also includes an index of disk blocks.

The index is arranged in a hierarchical manner. First, there are a few (e.g. 10) *direct* pointers, which list the first blocks of the file. Thus for small files all the necessary pointers are included in the inode, and once the inode is read into memory, they can all be found. As small files are much more common than large ones, this is efficient.

If the file is larger, so that the direct pointers are insufficient, the *indirect* pointer is used. This points to a whole block of additional direct pointers, which each point to a block of file data. The indirect block is only allocated if it is needed, i.e. if the file is bigger than 10 blocks. As an example, assume blocks are 1024 bytes (1 KB), and each pointer is 4 bytes. The 10 direct pointers then provide access to a maximum of 10 KB. The indirect block contains 256 additional pointers, for a total of 266 blocks (and 266 KB).

If the file is bigger than 266 blocks, the system resorts to using the *double* indirect pointer, which points to a whole block of indirect pointers, each of which point to an additional block of direct pointers. The double indirect block has 256 pointers to indirect blocks, so it represents a total of 65536 blocks. Using it, file sizes can grow to a bit over 64 MB. If even this is not enough, the *triple* indirect pointer is used. This points to a block of double indirect pointers.

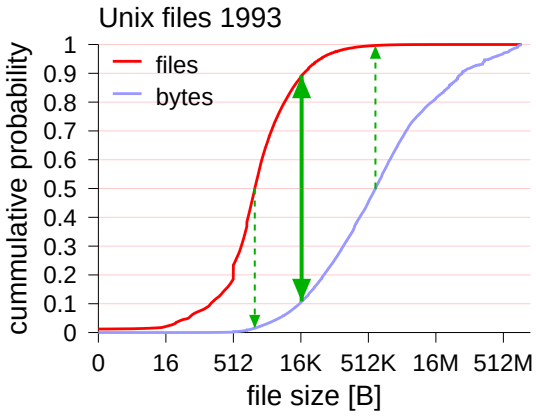


A nice property of this hierarchical structure is that the time needed to find a block is logarithmic in the file size. And note that due to the skewed structure, it is indeed logarithmic in the actual file size — not in the maximal supported file size. The extra levels are only used in large files, but avoided in small ones.

Exercise 141 *what file sizes are possible with the triple indirect block? what other constraints are there on file size?*

The Distribution of File Sizes

The distribution of file sizes is one of the examples of heavy-tailed distributions in computer workloads. The plot to the right shows the distribution of over 12 million files from over 1000 Unix file systems collected in 1993 [8]. Similar results are obtained for more modern file systems.

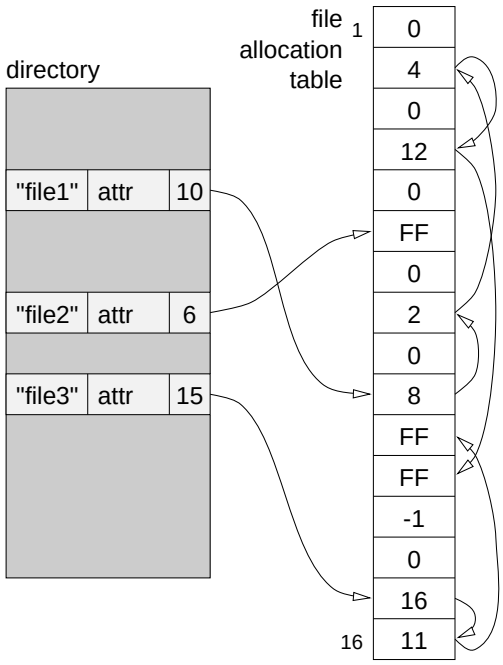


The distribution of files is the top line (this is a CDF, i.e. for each size x it shows the probability that a file will be no longer than x). For example, we can see that about 20% of the files are up to 512 bytes long. The bottom line is the distribution of bytes: for each file size x, it shows the probability that an arbitrary byte *belong to* a file no longer than x. For example, we can see that about 10% of the bytes belong to files that are up to 16 KB long.

The three vertical arrows allow us to characterize the distribution [4]. The middle one shows that this distribution has a "joint ratio" of 11/89. This means that the top 11% of the files are so big that together they account for 89% of the disk space. At the same time, the bottom 89% of the files are so small that together they account for only 11% of the disk space. The leftmost arrow shows that the bottom *half* of the files are so small that they only account for 1.5% of the disk space. The rightmost arrow shows that the other end of the distribution is even more extreme: half of the disk space is accounted for by only 0.3% of the files, which are each very big.

FAT uses a linked list

A different structure is used by FAT, the original DOS file system (which has the dubious distinction of having contributed to launching Bill Gates's career). FAT stands for "file allocation table", the main data structure used to allocate disk space. This table, which is kept at the beginning of the disk, contains an entry for each disk block (or, in more recent versions, each cluster of consecutive disk blocks that are allocated as a unit). Entries that constitute a file are linked to each other in a chain, with each entry holding the number of the next one. The last one is indicated by a special marking of all 1's (FF in the figure). Unused blocks are marked with 0, and bad blocks that should not be used also have a special marking (-1 in the figure).



Exercise 142 *Repeat Ex. 135 for the Unix inode and FAT structures: what happens if you seek beyond the current end of the file, and then write some data?*

Exercise 143 *What are the pros and cons of the Unix inode structure vs. the FAT structure? Hint: consider the distribution of file sizes shown above.*

> **FAT's Success and Legacy Problems**
> The FAT file system was originally designed for storing data on floppy disks. Two leading considerations where therefore simplicity and saving space. As a result file names were limited to the 8.3 format, where the name is no more than 8 characters, followed by an extension of 3 characters. In addition, the pointers were 2 bytes, so the table size was limited to 64K entries.
>
> The problem with this structure is that each table entry represents an allocation unit of disk space. For small disks it was possible to use an allocation unit of 512 bytes. In fact, this is OK for disks of up to $512 \times 64\text{K} = 32\text{MB}$. But when bigger disks became available, they had to be divided into the same 64K allocation units. As a result the allocation units grew considerably: for example, a 256MB disk was allocated in units of 4K. This led to inefficient disk space usage, because even small files had to allocate at least one unit. But the design was hard to change because so many systems and so much software were dependent on it.

6.4.2 Data Layout on the Disk

The structures described above allow one to find the blocks that were allocated to a file. But which blocks should be chosen for allocation? Obviously blocks can be chosen

at random — just take the first one you find that is free. But this can have adverse effects on performance, because accessing different disk blocks requires a physical movement of the disk head. Such physical movements are slow relative to a modern CPU — they can take milliseconds, which is equivalent to millions of instructions. This is why a process is blocked when it performs I/O operations. The mechanics of disk access and their effect on file system layout are further explained in Appendix C.

Placing related blocks together improves performance

The traditional Unix file system is composed of 3 parts: a superblock (which contains data about the size of the system and the location of free blocks), inodes, and data blocks. The conventional layout is as follows: the superblock is the first block on the disk, because it has to be at a predefined location[2]. Next come all the inodes — the system can know how many there are, because this number appears in the superblock. All the rest are data blocks.

The problem with this layout is that it entails much seeking. Consider the example of opening a file named `/a/b/c`. To do so, the file system must access the root inode, to find the blocks used to implement it. It then reads these blocks to find which inode has been allocated to directory `a`. It then has to read the inode for `a` to get to its blocks, read the blocks to find `b`, end so on. If all the inodes are concentrated at one end of the disk, while the blocks are dispersed throughout the disk, this means repeated seeking back and forth.

A possible solution is to try and put inodes and related blocks next to each other, in the same set of cylinders, rather than concentrating all the inodes in one place (a cylinder is a set of disk tracks with the same radius; see Appendix C). This was done in the Unix fast file system. However, such optimizations depend on the ability of the system to know the actual layout of data on the disk, which tends to be hidden by modern disk controllers [1]. Modern systems are therefore limited to using logically contiguous disk blocks, hoping that the disk controller indeed maps them to physically proximate locations on the disk surface.

Exercise 144 *The superblock contains the data about all the free blocks, so every time a new block is allocated we need to access the superblock. Does this entail a disk access and seek as well? How can this be avoided? What are the consequences?*

Log structured file systems reduce seeking

The use of a large buffer cache and aggressive prefetching can satisfy most read requests from memory, saving the overhead of a disk access. The next performance bottleneck is then the implementation of small writes, because they require much seeking to get to the right block. This can be solved by not writing the modified blocks

[2]Actually it is usually the second block — the first one is a boot block, but this is not part of the file system.

in place, but rather writing a single continuous log of all changes to all files and metadata. In addition to reducing seeking, this also improves performance because data will tend to be written sequentially, thus also making it easier to read at a high rate when needed.

Of course, this complicates the system's internal data structures. When a disk block is modified and written in the log, the file's inode needs to be modified to reflect the new location of the block. So the inode also has to be written to the log. But now the location of the inode has also changed, so this also has to be updated and recorded. To reduce overhead, metadata is not written to disk immediately every time it is modified, but only after some time or when a number of changes have accumulated. Thus some data loss is possible if the system crashes, which is the case anyway.

Another problem is that eventually the whole disk will be filled with the log, and no more writing will be possible. The solution is to perform garbage collection all the time: we write new log records at one end, and delete old ones at the other. In many cases, the old log data can simply be discarded, because it has since been overwritten and therefore exists somewhere else in the log. Pieces of data that are still valid are simply re-written at the end of the log.

To read more: Log structured file systems were introduced by Rosenblum and Ousterhout [12].

Logical volumes avoid disk size limitations

The discussion so far has implicitly assumed that there is enough space on the disk for the desired files, and even for the whole file system. With the growing size of data sets used by modern applications, this can be a problematic assumption. The solution is to use another layer of abstraction: logical volumes.

A logical volume is an abstraction of a disk. A file system is created on top of a logical volume, and uses its blocks to store metadata and data. In many cases, the logical volume is implemented by direct mapping to a physical disk or a disk partition (a part of the disk that is disjoint from other parts that are used for other purposes). But it is also possible to create a large logical volume out of several smaller disks. This just requires an additional level of indirection, which maps the logical volume blocks to the blocks of the underlying disks.

6.4.3 Reliability

By definition, the whole point of files to to store data *permanently*. This can run into two types of problems. First, the system may crash leaving the data structures on the disk in an inconsistent state. Second, the disks themselves sometimes fail. Luckily, this can be overcome.

Journaling improves reliability using transactions

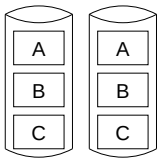
Surviving system crashes is done by journaling. This means that each modification of the file system is handled as a database transaction, implying all-or-nothing semantics.

The implementation involves the logging of all operations. First, a log entry describing the operation is written to disk. Then the actual modification is done. If the system crashes before the log entry is completely written, then the operation never happened. If it crashes during the modification itself, the file system can be salvaged using the log that was written earlier.

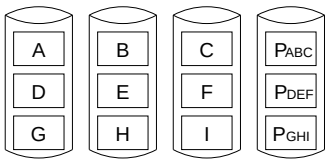
RAID improves reliability using redundancy

Reliability in the face of disk crashes can be improved by using an array of small disks rather than one large disk, and storing redundant data so that any one disk failure does not cause any data loss. This goes by the acronym RAID, for "redundant array of inexpensive (or independent) disks" [10]. There are several approaches:

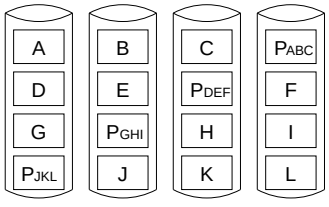
RAID 1: mirroring — there are two copies of each block on distinct disks. This allows for fast reading (you can access the less loaded copy), but wastes disk space and delays writing.



RAID 3: parity disk — data blocks are distributed among the disks in round-robin manner. For each set of blocks, a parity block is computed and stored on a separate disk. Thus if one of the original data blocks is lost due to a disk failure, its data can be reconstructed from the other blocks and the parity. This is based on the self-identifying nature of disk failures: the controller knows which disk has failed, so parity is good enough.



RAID 5: distributed parity — in RAID 3, the parity disk participates in every write operation (because this involves updating some block and the parity), and becomes a bottleneck. The solution is to store the parity blocks on different disks.



(There are also even numbers, but in retrospect they turned out to be less popular).

How are the above ideas used? One option is to implement the RAID as part of the file system: whenever a disk block is modified, the corresponding redundant block is updated as well. Another option is to buy a disk controller that does this for

you. The interface is the same as a single disk, but it is faster (because several disks are actually used in parallel) and is much more reliable. While such controllers are available even for PCs, they are still rather expensive.

To read more: The definitive survey of RAID technology was written by Chen and friends [3]. An interesting advanced system is the HP AutoRAID [13], which uses both RAID 1 and RAID 5 internally, moving data from one format to the other according to space availability and usage. Another interesting system is Zebra, which combines RAID with a log structured file system [6].

6.5 Summary

Abstractions

Files themselves are an abstraction quite distant from the underlying hardware capabilities. The hardware (disks) provides direct access to single blocks of a given size. The operating system builds and maintains the file system, including support for files of arbitrary size, and their naming within a hierarchical directory structure.

the operating system itself typically uses a lower-level abstraction of a disk, namely logical volumes.

Resource management

As files deal with permanent storage, there is not much scope for resource management — either the required space is available for exclusive long-term usage, or it is not. The only management in this respect is the enforcement of disk quotas.

Disk scheduling, if still practiced, has a degree of resource management.

Implementation

Files and directories are represented by a data structure with the relevant metadata; in Unix this is called an inode. Directories are implemented as files that contain the mapping from user-generated names to the respective inodes. Access to disk is mediated by a buffer cache.

Workload issues

Workload issues determine several aspects of file system implementation and tuning.

The distribution of file sizes determines what data structures are useful to store a file's block list. Given that a typical distribution includes multiple small files and few very large files, good data structures need to be hierarchical, like the Unix inode that can grow as needed by using blocks of indirect pointers. This also has an effect on the block size used: if it is too small disk access is less efficient, but if it is too big too much space is lost to fragmentation when storing small files.

Dynamic aspects of the workload, namely the access patterns, are also very important. The locality of data access and the fact that a lot of data is deleted or modified a short time after it is written justify (and even necessitate) the use of a buffer cache. The prevailing use of sequential access allows for prefetching, and also for optimizations of disk layout.

Hardware support

Hardware support exists at the I/O level, but not directly for files. One form of support is DMA, which allows slow I/O operations to be overlapped with other operations; without it, the whole idea of switching to another process while waiting for the disk to complete the transfer would be void.

Another aspect of hardware support is the migration of functions such as disk scheduling to the disk controller. This is actually implemented by firmware, but from the operating system point of view it is a hardware device that presents a simpler interface that need not be controlled directly.

Bibliography

[1] D. Anderson, "*You don't know Jack about disks*". *Queue* **1(4)**, pp. 20–30, Jun 2003.

[2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "*Measurements of a distributed file system*". In 13th *Symp. Operating Systems Principles*, pp. 198–212, Oct 1991. Correction in *Operating Systems Rev.* **27(1)**, pp. 7–10, Jan 1993.

[3] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "*RAID: high-performance, reliable secondary storage*". *ACM Comput. Surv.* **26(2)**, pp. 145–185, Jun 1994.

[4] D. G. Feitelson, "*Metrics for mass-count disparity*". In 14th *Modeling, Anal. & Simulation of Comput. & Telecomm. Syst.*, pp. 61–68, Sep 2006.

[5] E. Freeman and D. Gelernter, "*Lifestreams: a storage model for personal data*". *SIGMOD Record* **25(1)**, pp. 80–86, Mar 1996.

[6] J. H. Hartman and J. K. Ousterhout, "*The Zebra striped network file system*". *ACM Trans. Comput. Syst.* **13(3)**, pp. 274–310, Aug 1995.

[7] J. J. Hull and P. E. Hart, "*Toward zero-effort personal document management*". *Computer* **34(3)**, pp. 30–35, Mar 2001.

[8] G. Irlam, "*Unix file size survey - 1993*". URL http://www.gordoni.com/ufs93.html.

[9] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "*A trace-driven analysis of the UNIX 4.2 BSD file system*". In 10th *Symp. Operating Systems Principles*, pp. 15–24, Dec 1985.

[10] D. A. Patterson, G. Gibson, and R. H. Katz, "*A case for redundant arrays of inexpensive disks (RAID)*". In *SIGMOD Intl. Conf. Management of Data*, pp. 109–116, Jun 1988.

[11] J. Raskin, *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley, 2000.

[12] M. Rosenblum and J. K. Ousterhout, "*The design and implementation of a log-structured file system*". *ACM Trans. Comput. Syst.* **10(1)**, pp. 26–52, Feb 1992.

[13] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "*The HP AutoRAID hierarchical storage system*". *ACM Trans. Comput. Syst.* **14(1)**, pp. 108–136, Feb 1996.