Chapter 6

File Systems

General-purpose operating systems typically provide access to block storage (i.e. disks) via a *file system*, which provides a much more application- and user-friendly interface to storage. From the point of view of the user, a file system contains the following elements:

- a *name space*, the set of names identifying objects;
- *objects* such as the files themselves as well as directories and other supporting objects;
- *operations* on these objects.

Hierarchical namespace: File systems have traditionally used a tree-structured namespace[1], as shown Figure 6.1. This tree is constructed via the use of *directories*, or objects in the namespace which map strings to further file system objects. A full filename thus specifies a *path* from the root, through the tree, to the object (a file or directory) itself. (Hence the use of the term "path" to mean "filename" in Unix documentation)

File: Early operating systems supported many different file types—binary executables, text files, and record-structured files, and others. The Unix operating system is the earliest I know of that restricted files to sequences of 8-bit bytes; it is probably not a coincidence that Unix arrived at the same time as computers which dealt only with multiples of 8-bit bytes (e.g. 16 and 32-bit words), replacing older systems which frequently used odd word sizes such as 36 bits. (Note that a machine with 36-bit instructions already needs two incompatible types of files, one for text and one for executable code)

[1]Very early file systems sometimes had a single flat directory per user, or like MS-DOS 1.0, a single directory per floppy disk.

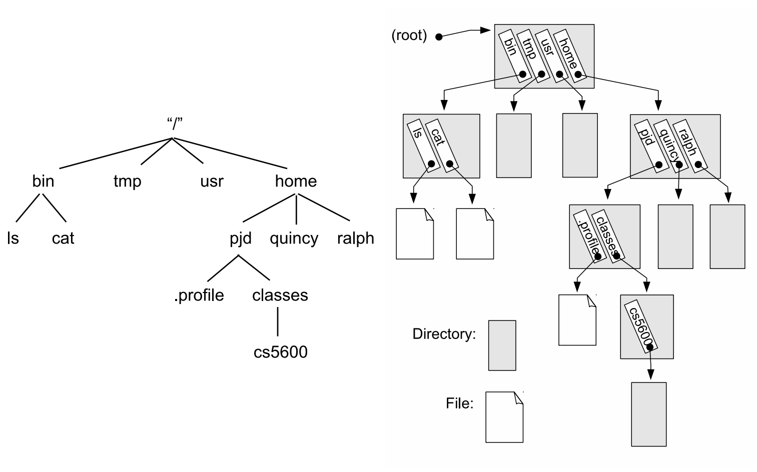


Figure 6.1: Logical view (left) and implementation (right) of a hierarchical file system name space.

Modern operating systems follow the UNIX model, which imposes no structure on a file—a file is merely a sequence of bytes.[2] Any structure to the file (such as a JPEG image, an executable program, or a database) is the responsibility of applications which read and write the file. The file format is commonly indicated by a file extension like .jpg or .xml, but this is just a convention followed by applications and users. You can do things like rename file.pdf to file.jpg, which will confuse some applications and users, but have no effect on the file contents.

Data in a byte-sequence file is identified by the combination of the file and its offset (in bytes) within the file. Unlike in-memory objects in an application, where a reference (pointer) to a component of an object may be passed around independently, a portion of a file cannot be named without identifying the file it is contained in. Data in a file can be created by a write which appends more data to the end of a shorter file, and modified by over-writing in the middle of a file. However, it can't be "moved" from one offset to another: if you use a text editor to add or delete text in the middle of a file, the editor must re-write the entire file (or at least from the modified part to the end).

Unix file name translation: each process has an associated *current di-*

[2]Almost. Apple OSX uses *resource forks* to store information associated with a file (HFS and HFS+ file systems only), Windows NTFS provides for multiple data streams in single file, although they were never put to use, and several file systems support *file attributes*, small tags associated with a file..

rectory, which may be changed via the `chdir` system call. File names beginning in '`/`' are termed *absolute* names, and are interpreted relative to the root of the naming tree, while *relative* names are interpreted beginning at the current directory. (In addition, `d/..` always points to the parent directory of `d`, and `d/.` points to `d` itself.) Thus in the file system in Figure 6.1, if the current directory were `/home`, the the paths `pjd/.profile` and `/home/pjd/.profile` refer to the same file, and `../bin/cat` and `/bin/cat` refer to the same file.

6.1 File System Operations:

There are several common types of file operations supported by Linux (and with slight differences, Windows). They can be classified into three main categories: open/close, read/write, and naming and directories.

Open/close: In order to access a file in Linux (or most operating systems) you first need to open the file, passing the file name and other parameters and receiving a *handle* (called a *file descriptor* in Unix) which may be used for further operations. The corresponding system calls are:

- `int desc = open(name, O_READ)` - Verify that file `name` exists and may be read, and then return a *descriptor* which may be used to refer to that file when reading it.
- `int desc = open(name, O_WRITE | flags, mode)` - Verify permissions and open `name` for writing, creating it (or erasing existing contents) if necessary as specified in `flags`. Returns a descriptor which may be used for writing to that file.
- `close(desc)` - stop using this descriptor, and free any resources allocated for it.

Note that application programs rarely use the system calls themselves to access files, but instead use higher-level frameworks, ranging from Unix Standard I/O to high-level application frameworks.

Read/Write operations: To get a file with data in it, you need to write it; to use that data you need to read it. To allow reading and writing in units of less than an entire file, or tedius calculations of the current file offset, UNIX uses the concept of a *current position* associated with a file descriptor. When you read 100 bytes (i.e. bytes 0 to 99) from a file this pointer advances by 100 bytes, so that the next read will start at byte 100, and similarly for write. When a file is opened for reading the pointer starts at 0; when open for writing the application writer can choose to start at the beginning (default) and overwrite old data, or start at the end (`O_APPEND` flag) to append new data to the file.

System calls for reading and writing are:

- `n = read(desc, buffer, max)` - Read `max` bytes (or fewer if the end of the file is reached) into `buffer`, starting at the current position, and returning the actual number of bytes `n` read; the current position is then incremented by `n`.
- `n = write(desc, buffer, len)` - write `len` bytes from `buffer` into the file, starting at the current position, and incrementing the current position by `len`.
- `lseek(desc, offset, flag)` Set an open file's current position to that specified by `offset` and `flag`, which specifies whether `offset` is relative to the beginning, end, or current position in the file.

Note that in the basic Unix interface (unlike e.g. Windows) there is no way to specify a particular location in a file to read or write from[3]. Programs like databases (e.g. SQLite, MySQL) which need to write to and read from arbitrary file locations must instead move the current position by using `lseek` before a read or write. However most programs either read or write a file from the beginning to the end (especially when written for an OS that makes it easier to do things that way), and thus don't really need to perform seeks. Because most Unix programs use simple "stream" input and output, these may be re-directed so that the same program can—without any special programming—read from or write to a terminal, a network connection, a file, or a pipe from or to another program.

Naming and Directories: In Unix there is a difference between a name (a directory entry) and the object (file or directory) that the name points to. The naming and directories operations are:

- `rename(path1, path2)` - Rename an object (i.e. file or directory) by either changing the name in its directory entry (if the destination is in the same directory) or creating a new entry and deleting the old one (if moving into a new directory).
- `link(path1, path2)` Add a *hard link* to a file[4].

[3]On Linux the `pread` and `pwrite` system calls allow specifying an offset for the read or write; other UNIX-derived operating systems have their own extensions for this purpose.

[4]A hard link is an additional directory entry pointing to the same file, giving the file two (or more) names. Hard links are peculiar to Unix, and in modern systems have mostly been replaced with symbolic links (covered next); however Apple's Time Machine makes very good use of them: multiple backups can point to the same single copy of an un-modified file using hard links.

- `unlink(path)` - Delete a file.[5]
- `desc = opendir(path)`
  `readdir(desc, dirent*), dirent=(name,type,length)`
  This interface allows a program to enumerate names in a directory, and determine their type. (i.e. file, directory, symbolic link, or special-purpose file)
- `stat(file, statbuf)`
  `fstat(desc, statbuf)` - returns file attributes - size, owner, permissions, modification time, etc. In Unix these are attributes of the file itself, residing in the i-node, and can't be found in the directory entry - otherwise it would be necessary to keep multiple copies consistent.
- `mkdir(path)`
  `rmdir(path)` - directory operations: create a new, empty directory, or delete an empty directory.

Review Questions

6.1.1. Directories in most file systems only contain pointers to files, not to other directories: *True / False*

6.1.2. Which one or more of the following scenarios could cause the contents of the 1000th byte in a file to either change or cease to exist?

a) The file is renamed
b) The file is deleted
c) Bytes 500 through 600 in the file are over-written
d) Bytes 900 through 1200 are over-written

6.1.3. For the read operation `read(handle, buffer, max)`, the range of bytes to be read from the file (e.g. bytes 100 through 199) is determined by which of the following? (more than one may apply)

a) The 'buffer' and 'max' arguments
b) The file handle current position and file length
c) The 'max' argument
d) bytes 0 through 'max'

[5]Sort of. If there are multiple hard links to a file, then this just removes one of them; the file isn't deleted until the last link is removed. Even then it might not be removed yet - on Unix, if you delete an open file it won't actually be removed until all open file handles are closed.. In general, deleting open files is a problem: while Unix solves the problem by deferring the actual delete, Windows solves it by protecting open files so that they cannot be deleted

Symbolic links

An alternative to hard links to allow multiple names for a file is a third file system object (in addition to files and directories), a *symbolic link*. This holds a text string which is interpreted as a “pointer” to another location in the file system. When the kernel is searching for a file and encounters a symbolic link, it substitutes this text into the current portion of the path, and continues the translation process.

Thus if we have:

```
directory: /usr/program-1.0.1
file:      /usr/program-1.0.1/file.txt
sym link: /usr/program-current -> "program-1.0.1"
```

and if the OS is looking up the file `/usr/program-current/file.txt`, it will:

1. look up `usr` in the root directory, finding a pointer to the `/usr` directory
2. look up `program-current` in `/usr`, finding the link with contents `program-1.0.1`
3. look up `program-1.0.1` and use this result instead of the result from looking up `program-current`, getting a pointer to the `/usr/program-1.0.1` directory.
4. look up `file.txt` in this directory, and find it.

Note that unlike hard links, a symbolic link may be “broken”—i.e. if the file it points to does not exist. This can happen if the link was created in error, or the file or directory it points to is deleted later. In that case path translation will fail with an error:

```
pjd-1:tmp pjd$ ln -s /bad/file/name bad-link
pjd-1:tmp pjd$ ls -l bad-link
lrwxr-xr-x 1 pjd wheel 22 Aug 2 00:07 bad-link -> /bad/file/name
pjd-1:tmp pjd$ cat bad-link
cat: bad-link: No such file or directory
```

Finally, to prevent loops there is a limit on how many levels of symbolic link may be traversed in a single path translation:

```
pjd@pjd-fx:/tmp$ ln -s loopy loopy
pjd@pjd-fx:/tmp$ ls -l loopy
lrwxrwxrwx 1 pjd pjd 5 Aug 24 04:25 loopy -> loopy
pjd@pjd-fx:/tmp$ cat loopy
cat: loopy: Too many levels of symbolic links
pjd@pjd-fx:/tmp$
```

In early versions of Linux (pre-2.6.18) the link translation code was recursive, and this limit was set to 5 to avoid stack overflow. Current versions use an iterative algorithm, and the limit is set to 40.

Device Names vs. Mounting: A typical system may provide access to several file systems at once, e.g. a local disk and an external USB drive or network volume. In order to unambiguously specify a file we thus need to both identify the file within possibly nested directories in a single file system, as well as identifying the file system itself. (in Unix this name is called an *absolute pathname*, providing an unambiguous "path" to the file.) There are two common approaches to identifying file systems:

- Explicitly: each file system is given a name, so that a full pathname looks like e.g. `C:\MyDirectory\file.txt` (Windows[6]) or `DISK1:[MYDIR]file.txt` (VMS).
- Implicitly: a file system is transparently *mounted* onto a directory in another file system, giving a single uniform namespace; thus on a Linux system with a separate disk for user directories, the file "/etc/passwd" would be on one file system (e.g. "disk1"), while "/home/pjd/file.txt" would be on another (e.g. "disk2").

The actual implementation of mounting in Linux and other Unix-like systems is implemented via a *mount table*, a small table in the kernel mapping directories to directories on other file systems. In the example above, one entry would map "/home" on disk1 to ("disk2", "/"). As the kernel translates a pathname it checks each directory in this table; if found, it substitutes the mapped file system and directory before searching for an entry. Thus before searching "/home" on disk1 (which is probably empty) for the entry "pjd", the kernel will substitute the top-level directory on disk2,and then search for "pjd".

For a more thorough explanation of path translation in Linux and other Unix systems see the `path_resolution(7)` man page, which may be accessed with the command `man path_resolution`.

Review Questions

6.1.1. Creating, modifying, and deleting directories is performed by different system calls than creating and deleting files. Which of the following are possible reasons for this?

a) When deleting a directory, the OS must check to be sure that it is empty
b) Directories use a different kind of name from files
c) To prevent users from modifying directory data which is accessed by kernel code.

[6]Modern Windows systems actually use a mount-like naming convention internally; e.g. the `C:` drive actually corresponds to the name `\DosDevices\C:` in this internal namespace.

6.1.2. Which one of the following statements best describes the Unix mount table?

a) Used at startup to determine how to name different filesystems
b) A table in the kernel used to recognize where one filesystem is "attached" to another

6.2 File System Layout

To store a file system on a real disk, the high-level objects (directories, files, symbolic links) must be translated into fixed-sized blocks identified by logical block addresses.

Note that instead of 512-byte sectors, file systems traditionally use *disk blocks*, which are some small power-of-two multiple of the sector size, typically 1KB, 2KB, or 4KB. Reading and writing is performed in units of complete blocks, and addresses are stored as disk block numbers rather than LBAs, and are then multiplied by the appropriate value before being passed to the disk. Since modern disk drives have an internal sector size of 4 KB (despite pretending to support 512-byte sectors) and the virtual memory page size is 4 KB on most systems today, that has become a very common file system block size.

Designing on-disk data structures is complicated by the fact that for various reasons (virtual memory, disk controller restrictions, etc.) the data in a file needs to be stored in full disk blocks — e.g. bytes 0 through 4095 of a file should be stored in a single 4096-byte block. (This is unlike in-memory structures, where odd-sized allocations usually aren't a problem.)

In this section we examine a number of different file systems; we can categorize them by the different solutions their designers have come up with for the following three problems:

1. How to find objects (files, directories): file identification.
2. How to find the data within a file: file organization.
3. How to allocate free space for creating new files.

CD-ROM File System

In Figure 6.2 we see an example of an extremely simple file system, similar to early versions of the ISO-9660 file system for CD-ROM disks. Objects on disk are either files or directories, each composed of one or more 2048-

byte[7] *blocks*; all pointers in the file system are in terms of *block numbers*, with blocks numbered from block 0 at the beginning of the disk.

There are no links—each object has exactly one name—and the type of an object is indicated in its directory entry. (The only exception is the root directory, which has no name; however it is always found at the beginning of the disk) Finally, all objects are *contiguous*, allowing them to be identified by a starting block number and a length.

This organization is both compact and fairly efficient. As in almost all file systems, an object is located by using linear search to find each path component in the corresponding directory. Once a file is located, access to any position is straightforward and can be calculated from the starting block address of the file, as all files are contiguous.

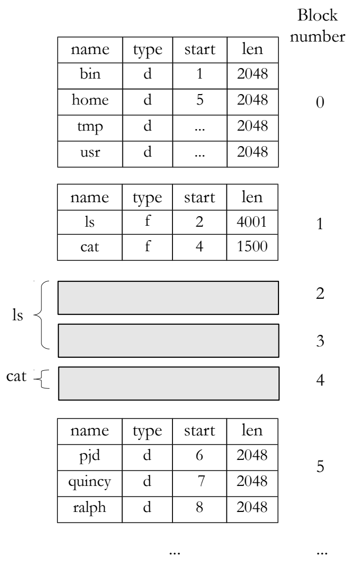


Figure 6.2: Simplified ISO-9660 (CD-ROM) file layout for tree in Figure 6.1, 2KB blocks

Contiguous organization works fine for a read-only file system, where all files (and their sizes) are available when the file system is created. It works poorly for writable file systems, however, as space would quickly fragment making it impossible to create large files. (Also the CDROM file system has no method for tracking free space, so allocation would be very inefficient.)

In the simple CD-ROM file system, what were the solutions to the three design problems?

1. File identification: files are identified by their starting block number
2. File organization: blocks in a file are contiguous, so an offset in the file can be found by adding to the starting block number.
3. Free space allocation: since it's a read-only file system, there is no free space to worry about.

[7] Why 2048? Because the designers of the CDROM file system defined it that way. Data is stored on CD in 2048-byte blocks plus error correction, making use of smaller block sizes difficult, and the authors evidently didn't see any need to allow larger block sizes, either.

Review Questions

6.2.1. On-disk structures must be constructed of disk blocks, rather than arbitrary-sized regions: *True / False*

6.2.2. A file system can use large blocks for the large files in a directory and small blocks for the small files: *True / False*

6.2.3. Not counting blocks used for the directory, how much space would be required to store 20 files, each 100 bytes long, in the CD-ROM file system described?

a) It would require 20 2048-byte blocks
b) It would require a single 2048-byte block

6.2.4. The CD-ROM file system described in this chapter tracks free space in its directories: *True / False*

MS-DOS file system

The next file system we consider is the MS-DOS (or FAT, File Allocation Table) file system. Here blocks within a file are organized in a linked list; however implementation of this list is somewhat restricted by the requirement that all access to the disk be done in multiples of a fixed block size.[8] Instead these pointers are kept in a separate array, with an entry corresponding to each disk block, in what is called the File Allocation Table.

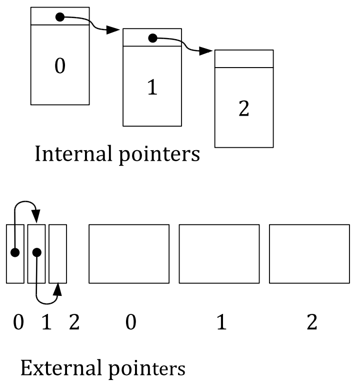


Figure 6.3: Linked list organization with in-object pointers (typical for in-memory structures) and external pointers, as used in MS-DOS File Allocation Table.

Entries in this table can indicate (a) the number of the next block in the file or directory, (b) that the block is the last one in a file or directory, or (c) the block is free. The FAT is thus used for free space management as well as file organization; when a block is needed the table may be searched for a free entry which can then be allocated.

Again, what were the solutions to the three design problems?

1. File identification - Files and directories are identified by their starting block number
2. File organization - blocks within a file are linked by pointers in the FAT
3. Free space allocation - free blocks are marked in the FAT, and linear search is used to find free space

Directories are similar to the CD-ROM file system - each entry has a name, the object type (file or directory), its length, and the starting address of the file contents. Note that although the last block of a file can be identified by a flag in the FAT, the length field is not redundant as it is still needed to know how much of the last block is valid. (E.g. a 5-byte file will require

[8]The astute reader will note that the pointer could use bytes within a block, causing each block to store slightly less than a full block of data. This would pose difficulties for operating systems such as Linux which tightly couple the virtual memory and file systems, and assume that each 4 KB virtual memory page corresponds to one (or maybe 2 or 4) file system blocks.

an entire block, but will only use 5 bytes in that block.) Sequential access to a file incurs overhead to fetch file allocation table entries, although since these are frequently used they may be cached; random access to a file, however, requires walking the linked list to find the corresponding entry, which can be slow even when cached in memory. (Consider random I/O within a 1 GB virtual disk image with 4 KB blocks—the linked list will be 256K long, and on average each I/O will require searching halfway through the list[9]).

Directories in the MS-DOS file system are similar to those in ISO-9660. Each directory entry is a fixed size and has a field indicating whether it is valid; to delete a file, this field is set to invalid and the blocks in that file are marked as free in the file allocation table. Only a single name per file is supported, and all file metadata (e.g. timestamps, permissions) is stored in the directory entry along with the size and first block number.

Like most file systems, linear search is used to locate a file in a directory. This is usually reasonably efficient (it's used by most Unix file systems, too) but works poorly for very large directories. (That's why your browser cache has filenames that look like `ab/xy/abxy123x.dat`, instead of putting all its files in the same directory.)

A note for the reader - the original MS-DOS file system only supported 8-byte upper-case names with 3-byte extensions, with (seemingly) no way to get around this restriction, since the size of a directory entry is fixed. A crazy mechanism was devised that is still used today: multiple directory entries are used for each file, with the extra entries filled with up to 13 2-character Unicode filename characters in not only the filename field, but also the space that would have otherwise been used for timestamp, size, starting block number, etc., and marked in a way that would be ignored by older versions of MS-DOS.

Review Questions

6.2.1. The MS-DOS file system identifies the blocks in a file through which of the following processes?

a) By marking them with the file ID in the file allocation table
b) By linking them with pointers at the beginning of each block
c) By linking them with pointers in the FAT

6.2.2. Which of these are differences between ext2 and the MS-DOS file system described previously?

[9] A benchmark run on login.ccs.neu.edu indicates that "pointer chasing" on a high-end Xeon takes about 200 ns when data is not in cache; each such random I/O would thus take about 25 ms of CPU time.

a) ext2 allows multiple names for the same file, while MS-DOS only allows one.
b) ext2 has less overhead than the MS-DOS file system.

Unix file systems (e.g. ext2)

File systems derived from the original Unix file system (e.g. Linux ext2 and ext3) use a per-file structure called an inode ("indirect node") designed with three goals in mind: (a) low overhead for small files, in terms of both disk seeks and allocated blocks[10], (b) ability to represent sufficiently large files without excessive storage space or performance overhead, and (c) crash resiliency—crashing while the file is growing should not endanger existing data.

Why not use e.g. a balanced binary tree? The in-memory tree structures from your algorithms class aren't appropriate for a file system, for several reasons: (a) the minimum allocation unit is a disk block, typically 4 KB, (b) disk seeks are really expensive, and (c) we want to avoid re-arranging existing data on disk as the file grows, so that we don't lose it if the system crashes mid-operation.

To do this, the inode uses an asymmetric tree, or actually a series of trees of increasing height with the root of each tree stored in the inode. As seen in Figure 6.4 the inode contains N *direct* block pointers (12 in ext2/ext3), so that files of N blocks or less need no indirect blocks. A single *indirect pointer* specifies an *indirect block*, holding pointers to blocks $N, N+1, ...N+N_1-1$ where N_1 is the number of block numbers that fit in a file system block (1024 for ext2 with a 4 KB blocksize). If necessary, the *double-indirect pointer* specifies a block holding pointers to N_1 indirect blocks, which in turn hold pointers to blocks $N+N_1...N+N_1+N_1^2-1$— i.e. an N_1-ary tree of height 2; a triple indirect block in turn points to a tree of height 3. For ext2 with 4-byte block numbers, if we use 4K blocks this gives a maximum file size of $(4096/4)^3$ 4 KiB[11] blocks, or 4.004 TiB. This organization allows random access within a file with overhead $O(logN)$ where N is the file size, which is vastly better than the $O(N)$ overhead of the MS-DOS File Access Table system.

In addition to the block pointers, the inode holds file *metadata* such as the owner, permissions, size, and timestamps. The separation of name (i.e. directory entry) and object (the inode and the blocks it points to) also allows files to have multiple names, which for historical reasons are called

[10]The median file size in a recent study was 4 KB, or one block

[11]When we're being really precise, we'll use KiB, MiB, GiB etc. to mean 2^{10}, 2^{20}, 2^{30} and KB, MB, GB to mean 10^3, 10^6 and 10^9.

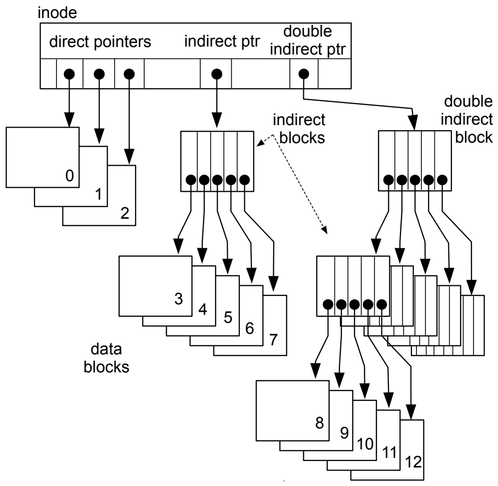


Figure 6.4: Inode-type file organization as found in many Unix file systems (e.g. Linux ext2, ext3). Note that the degree of branching is far lower than in real file systems, and the triple-indirect pointer is missing.

hard links. For the longest time hard links were a little-used capability of Unix-style file systems; however Apple Time Machine for the HFS+ file system makes good use of them to create multiple backup snapshots which share identical files to save space.

Since files can have multiple names, the inode also contains a reference count; as each name is deleted (via the `unlink` system call) the count is decremented, and when the count goes to zero the file is deleted. This also allows a file to have *zero* names—when a file is opened the reference count (in memory, not on disk) is incremented, and decremented when it is closed, so if you unlink a file which is in use, it is not actually deleted until the last open file descriptor is closed[12].

Ext2 space allocation: The original Unix file system used a free list to store a list of unused blocks; blocks were allocated from the head of this list for new files, and returned to the head when freed. As files were created and deleted this list became randomized, so that blocks allocated for a file were rarely sequential and disk seeks were needed for nearly every block read from or written to disk. This wasn't a significant problem, because

[12]Deleting open files is a tricky problem, as there's no good way to handle operations on those open handles after the file is deleted. Unix solves it by postponing the actual deletion until the file descriptor is closed; Windows instead locks the file against deletion until any open file handles are closed.

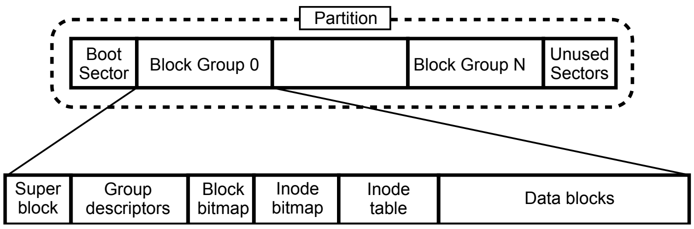


Figure 6.5: Ext2 on-disk layout

early Unix systems ran on machines with fast disks and excruciatingly slow CPUs. As computers got faster and users started noticing that the file system was horribly slow, the Fast File System (FFS) from Berkeley Unix replaced the free list with a more efficient mechanism, the *allocation bitmap*.

Ext2 is essentially a copy of FFS, and uses this bitmap mechanism. It keeeps a boolean array with one bit for each disk block; if the block is allocated the corresponding bit is set to '1', and cleared to '0' if it is freed. To allocate a block you read a portion of this bitmap into memory and scan for a '0' bit, changing it to '1' and writing it back. When you extend a file you begin the search at the bit corresponding to the last block in the file; in this way if a sequential block is available it will be allocated. This method results in files being allocated in a mostly sequential fashion, reducing disk seeks and greatly improving performance. (An additional bitmap is used for allocating inodes; in this case we don't care about sequential allocation, but it's a compact representation, and we can re-use some of the code written for block allocation.)

Block groups, as shown in Figure 6.5, are an additional optimization from FFS. Each block group is a miniature file system, with block and inode bitmaps, inodes, and data blocks. The file system tries to keep the inode and data blocks of a file in the same block group, as well as a directory and its contents. In this way common operations (e.g. open and read a file, or 'ls -l') will typically access blocks within a single block group, avoiding long disk seeks.

Long file names: Ext2 supports long file names using the mechanism used in FFS. Rather than treating the directory as an array of fixed-sized structures, it is instead organized as a sequence of length/value-encoded entries, with free space treated as just another type of entry. Directory search is performed using linear search.

Ext2 solutions to the three design problems?

1. File identification - files and directories are identified by inode number, and the location of the fixed-sized inode can be calculated from inode number and the inode table location.
2. File organization - blocks within a file are located via pointers from the inode
3. Free space allocation - free blocks are tracked in a free-space bitmap, and block groups are used to keep blocks from the same file near to each other, their inode, and their directory.

Note the difference here between the data structure (a bitmap) and strategies used such as trying to allocate the block immediately after the previous one written. The MS-DOS file system organizes its free list in an array, as well, and most of the allocation techniques introduced in the Berkeley Unix file system could be used with it. In practice, however, the MS-DOS file system was typically implemented with simple allocation strategies that resulted in significant file system fragmentation.

An additional anti-fragmentation strategy used by many modern operating systems is the enforcement of a maximum utilization, typically 90% or 95%, as when a file system is almost full, it is likely that any free space will be found in small fragments scattered throughout the disk. By limiting utilization to e.g. 90%—i.e. one block out of ten is free—we significantly increase the chance of finding multiple contiguous blocks when writing to a file, while greatly decreasing the fraction of the bitmap we may need to search to find a free block.

6.3 Superblock

Before a disk can be used in most systems it needs to be *initialized* or *formatted*—the basic file system structures need to be put in place, describing a file system with a single directory and no files. A key structure written in this process is the *superblock*, written at a well-known location on the disk. (This is often block 1, allowing block 0 to be used by the boot loader.) The superblock specifies various file system parameters, such as:

- Block size - most file systems can be formatted with different block sizes, and the OS needs to know this size before it can interpret any pointers given in terms of disk blocks. Historically larger blocks were used for performance and to allow larger file systems, and smaller blocks for space efficiency. In recent years disk drives have transitioned to using an internal block size of 4KB, while keeping the traditional 512-byte sector addressing, so any file system should use a block size of at least 4KB.

- Version - including a version number allows backwards compatibility as a file system evolves. That way you can upgrade your OS, for instance, without reformatting your disk.
- Other parameters - in the MS-DOS file system the OS needs to know how large the FAT table is, so that it doesn't accidently go off the end and start looking at the first data block. In ext2 you need to know the sizes of the block groups, as well as the bitmap sizes, how many inodes are in each group, etc.
- Dirty flag - when a file system is mounted, this flag is set; as part of a clean shutdown the flag is cleared again. If the system crashes without clearing the flag, at the next boot this indicates that additional error checks are needed before mounting the file system.

6.4 Extents, NTFS, and Ext4

The ext2 and MS-DOS file systems use separate pointers to every data block in a file, located in inodes and indirect blocks in the case of ext2, and in the file allocation table in MS-DOS. But the values stored in these pointers are often very predictable, because the file system attempts to allocate blocks sequentially to avoid disk seeks—if the first block in a file is block 100, it's highly likely that the second will be 101, the third 102, etc.

We can take advantage of this to greatly compress the information needed to identify the blocks in a file - rather than having separate pointers to blocks 100,101,...120 we just need to identify the starting block (100) and the length (21 blocks). This is shown in Figure 6.6, where five data blocks are identified by inodes or indirect block pointers; to the right, the same five data blocks are identified by a single extent. Why would we want to compress the information needed to organize the blocks in a file? Mostly for performance—although the code is more complicated, it will require fewer disk seeks to read from disk.

This organization is the basis of *extent-based* file systems, where blocks in a file are identified via one or more *extents*, or (start,length) pairs. The inode (or equivalent) can contain space for a small number of extents; if the file grows too big, then you add the equivalent of indirect blocks - extents pointing to blocks holding more extents. Both Microsoft NTFS and Linux ext4 use this sort of extent structure.

NTFS: Each NTFS file system has a Master File Table (MFT), which is somewhat like the inode table in ext2—each file or directory has an entry in this table which holds things like permissions, timestamps, and block information. (The superblock contains a pointer to the start of the MFT;

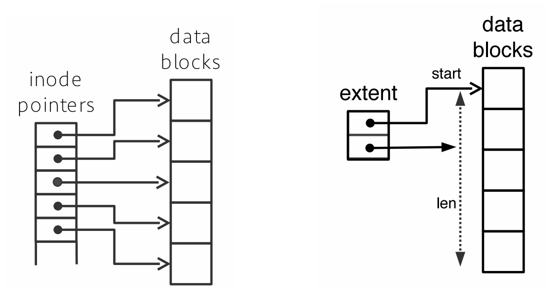


Figure 6.6: File structure—pointers vs. extents

the first entry in the MFT describes the MFT itself, so that it can grow as needed.) Each entry in the MFT is structured as a set of attributes, with a $Data attribute specifying the file contents. This attribute can be of two types: internal, where the attribute holds file data directly (for very small files), or external, in which case the $Data attribute holds a list of *extents*, or contiguous regions identified by a starting block and length.

If the number of extents grows too large to fit into the MFT entry, an $ATTRIBUTE_LIST field is added, holding a list of extents describing the blocks holding the list of extents describing the file. This can continue for one more level, which is enough to support files up to 16TB. Note that the amount of space taken by the $Data attribute depends not only on the size of the file, but its fragmentation; a very large file created on an empty file system might consist of only a few extents, while a modest-sized file created slowly (e.g. a log file) on a full file system might be composed of hundreds of extents.

Free space is handled similarly, as a list of extents sorted by starting block number; this allows the free space list to be easily compacted when storage is freed. (i.e. just by checking to see if it can be combined with its neighbors on either side) This organization makes it easy to minimize file fragmentation, reducing the number of disk seeks required to read a file or directory. It has the disadvantage that random file access is somewhat more complex, and appears to require reading the entire extent list to find which extent an offset may be found in. (A more complex organization could in fact reduce this overhead; however in practice it does not seem significant, as unless highly fragmented the extent lists tend to be fairly short and easily cached.)

NTFS solutions to the three design problems?

1. file identification - Master File Table entry

2. file organization - (possibly multi-level) extent list
3. free space management - sorted extent list.

Ext4: Ext4 supports extent-based file organization with minimal change to the inode structure in ext2/ext3: an extent tree is used, with each node explicitly marked as an interior or leaf node, as shown in Figure 6.7. The inode holds a four-entry extent tree node, allowing small files to be accessed without additional lookup steps, while for moderate-sized files only a single level of the tree (a “leaf node” in the figure) is needed.

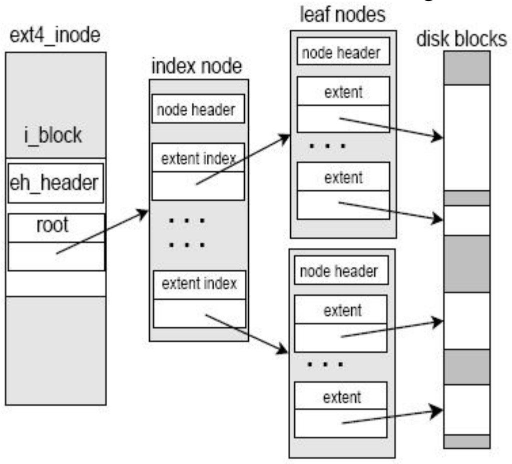


Figure 6.7: Ext4 on-disk structure

6.5 Smarter Directories

In the CD-ROM, MS-DOS, and ext2 file systems, a directory is just an array of directory entries, in unsorted order. To find a file, you search through the directory linearly; to delete a file, you mark its entry as unused; finally, to create a new entry, you find any entry that’s free. (It’s a bit more complicated for file systems like ext2 which have variable-length directory entries, but not much.)

From your data structures class you should realize that linear search isn’t an optimal data structure for searching, but it’s simple, robust, and fast enough for small directories, where the primary cost is retrieving a block of data from the disk. As an example, one of my Linux machines has 94944 directories that use a single 4KB block, another 957 that use 2 to 5 blocks, and only 125 larger than 5 blocks. In other words, for the 99% of the directories that fit within a single 4 KB block, a more complex algorithm would not reduce the amount of data read from disk, and the difference between $O(N)$ and $O(logN)$ algorithms when searching a single block is negligible.

However the largest directories are actually quite big: the largest on this machine, for example, has 13,748 entries; another system I measured had a database directory containing about 64,000 files with long file names, or roughly 4000 blocks (16 MB) of directory data. Since directories tend to grow slowly, these blocks were probably allocated a few at a time, resulting in hundreds or thousands of disk seeks to read the entire directory into memory. At 15 ms per seek, this could require 10-30 seconds or more, and once the data was cached in memory, linear search in a 16 MB array will probably take a millisecond or two.

To allow directories with tens of thousands of files or more, modern file systems tend to use more advanced data structures for their directories. NTFS (and Linux Btrfs) use B-trees, a form of a balanced tree. Other file systems, like Sun ZFS, use hash tables for their directories, while ext4 uses a hybrid hash/tree structure. If you're really interested, you can look these up in Google.

6.6 The B-tree

The B-tree is one of those widely-used data structures that you never see in your data structures course. It's not a file system— the B-tree is a disk-optimized search structure, optimized for the case where accessing a block of information is much more expensive (e.g. requiring a disk seek) than searching through that block after it has been accessed. It has been used for file systems, databases, and similar purposes since the 1970s, along with various extensions (e.g. B^+-trees) which are not described here.

B-tries are balanced trees made up of large blocks, with a high branching factor, in order to reduce the number of block accesses needed for an operation. Interior and leaf nodes are identical; each contains a sorted list of key/-value pairs, and (in non-leaf nodes) pointers between pairs of keys, pointing to subtrees holding keys which are between those two values. The tree grows from the bottom up: if a block overflows, you split it, dividing the contents between two blocks, and add a pointer to the new block in the correct position in the parent; if the parent overflows it is split, and so on. If the root node splits, a new root is allocated with pointers to the two pieces.

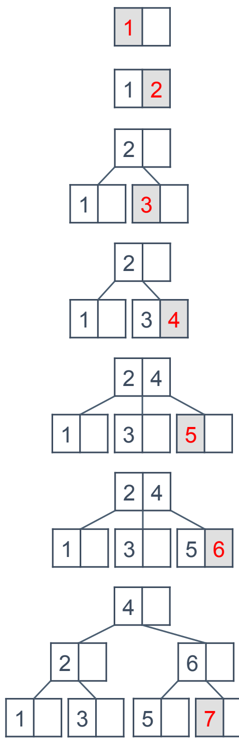


Figure 6.8: B-tree growth

If the branching factor of a B-tree is m, then each block (except for the root) holds between m/2 and m entries. In the example shown in Figure 6.8, m=2; in a real system each node would contain many more entries.

In Figure 6.8 we see seven values being added to the tree, which grows "from the bottom up":

1. The first value goes in the root
2. Since the root isn't full, the second value goes here too
3. Now it's full - split the block. Since the block doesn't have a parent (it's the root) we add one, which becomes the new root
4. '4' fits into one of the leaf nodes where there's room
5. '5' doesn't fit, so we split the node. There's room in the parent to hold another pointer
6. '6' fits in the leaf node

7. ’7’ doesn’t, so we split the leaf node, but that causes the parent node to overflow, so we split it, and have to add a new parent node which becomes the new root.

6.7 Consistency and Journaling

Unlike in-memory structures, data structures on disk must survive system crashes, whether due to hardware reasons (e.g. power failure) or software failures. This is a different problem than the consistency issues we dealt with for in-memory structures, where data corruption could only occur due to the action of other threads, and could be prevented by the proper use of mutexes and similar mechanisms. Unfortunately there is no mutex which will prevent a system from crashing before the mutex is unlocked, or file system designers would use it liberally. The problem is compounded by the fact that operating systems typically cache reads and writes to increase performance, so that writes to the disk may occur in a much different order than that in which they were issued by the file system code.

In its simplest form the problem is that file system operations often involve writing to multiple disk blocks—for example, moving a file from one directory to another requires writing to blocks in the source and destination directories, while creating a file writes to the block and inode allocation bitmaps, the new inode, the directory block, and the file data block or blocks[13]. If some but not all of these writes occur before a crash, the file system may become *inconsistent*—i.e. in a state not achievable through any legal sequence of file system operations, where some operations may return improper data or cause data loss.

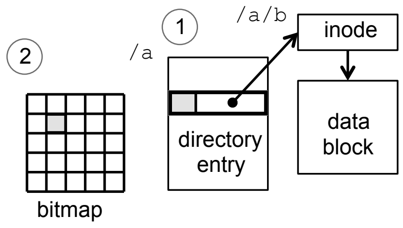


Figure 6.9: File, directory, bitmap

For a particularly vicious example, consider deleting the file /a/b as shown in Figure 6.9, which requires the following actions:

1. Clear the directory entry for /a/b. This is done by marking the entry as unused and writing its block back to the directory.
2. Free the file data block, by clearing the corresponding entry in the block allocation bitmap

[13]These steps ignore inode writes to update file or directory modification times.

This results in two disk blocks being modified and written back to disk; if the blocks are cached and written back at a later point in time they may be written to disk in any order. (this doesn't matter for running programs, as when they access the file system the OS will check cached data before going to disk)

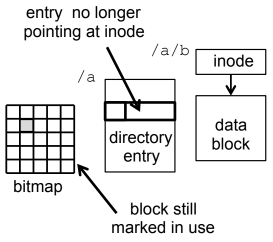


Figure 6.10: Directory block written before crash

If the system crashes (e.g. due to a power failure) after one of these blocks has been written to disk, but not the other, two case are possible:

1. The directory block is written, but not the bitmap. The file is no longer accessible, but the block is still marked as in use. This is a disk space leak (like a memory leak), resulting in a small loss of disk space but no serious problems.
2. The bitmap block is written, but not the directory. Applications are still able to find the file, open it, and write to it, but the block is also available to be allocated to a new file or directory. This is much more serious.

If the same block is now re-allocated for a new file (`/a/c` in this case) we now have two files sharing the same data block, which is obviously a problem. If an application writes to `/a/b` it will also overwrite any data in `/a/c`, and vice versa. If `/a/c` is a directory rather than a file things are even worse - a write to `/a/b` will wipe out directory entries, causing files pointed to by those entries to be lost. (The files themselves won't be erased, but without directory entries pointing to them there won't be any way for a program to access them.)

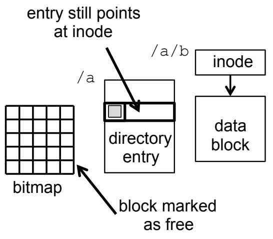


Figure 6.11: Bitmap block written before crash

This can be prevented by writing blocks in a specific order—for instance in this case the directory entry could always be cleared before the block is marked as free, so that in the worst case a crash might cause a few data blocks to become unusable. Unfortunately this is very slow, as these writes must be done synchronously, waiting for each write to complete before issuing the next one.

Fsck / chkdsk: One way to prevent this is to run a disk checking routine every time the system boots after a crash. The dirty flag in the file system superblock was described in the section above; when a machine boots, if the file system is marked dirty, (`fsck`, or `chkdsk` in Windows) is run to repair any problems.

In particular, the Unix file system checker performs the following checks and corrections:

1. Blocks and sizes. Each allocated inode is checked to see that (a) the number of blocks reachable through direct and indirect pointers is consistent with the file size in the inode, (b) all block pointers are within the valid range for the volume, and (c) no blocks are referenced by more than one inode.
2. Pathnames. The directory tree is traversed from the root, and each entry is checked to make sure that it points to a valid inode of the same type (directory / file / device) as indicated in the entry.
3. Connectivity. Verifies that all directory inodes are reachable from the root.
4. Reference counts. Each inode holds a count of how many directory entries (hard links) are pointing to it. This step validates that count against the count determined by traversing the directory tree, and fixes it if necessary.
5. "Cylinder Groups" The block and inode bitmaps are checked for consistency. In particular, are all blocks and inodes reachable from the root marked in use, and all unreachable ones marked free?
6. "Salvage Cylinder Groups" Free inode and block bitmaps are updated to fix any discrepancies.

This is a lot of work, and involves a huge number of disk seeks. On a large volume it can take hours to run. Note that full recovery may involve a lot of manual work; for instance, if fsck finds any files without matching directory entries, it puts them into a `lost+found` directory with numeric names, leaving a human (i.e. you) to figure out what they are and where they belong.

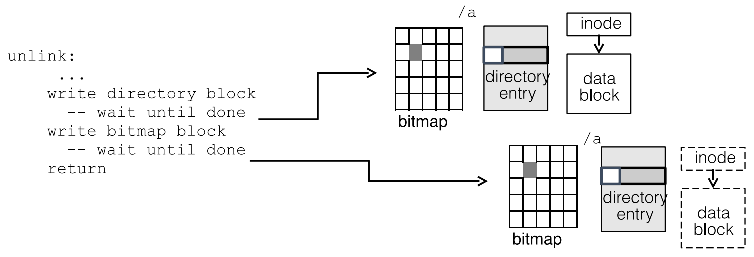
Checking disks at startup worked fine when disks were small, but as they got larger (and seek times didn't get faster) it started taking longer and longer to check a file system after a crash. Uninterruptible power supplies help, but not completely, since many crashes are due to software faults in the operating system. The corruption problem you saw was due to inconsistency in the on-disk file system state. In this example, the free space bitmap did not agree with the directory entry and inode. If the file system can ensure that the on-disk data is always in a consistent state, then it should be possible to prevent losing any data except that being written at the exact moment of the crash.

Performing disk operations synchronously (and carefully ordering them in the code) will prevent inconsistency, but as described above imposes excessive performance costs. Instead a newer generation of file systems, termed *journaling* file systems, has incorporated mechanisms which add additional information which can be used for recovery, allowing caching and efficient use of the disk, while maintaining a consistent on-disk state.

6.8 Journaling

Most modern file systems (NTFS, ext3, ext4, and various others) use *journaling*, a variant of the database technique of *write-ahead logging*. The idea is to keep a log which records the changes that are going to be made to the file system, *before those changes are made*. After an entry is written to the log, the changes can be written back in any order; after they are all written, the section of log recording those changes can be freed.

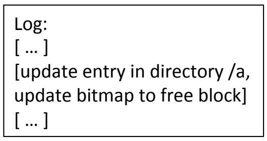
When recovering from a crash, the OS goes through the log and checks that all the changes recorded there have been performed on the file system itself[14]. Some thought should convince you that if a log entry is written, then the modification is guaranteed to happen, either before or after a crash; if the log entry isn't written completely then the modification never happened. (There are several ways to detect a half-written log entry, including using an explicit end marker or a checksum; we'll just assume that it's possible.)



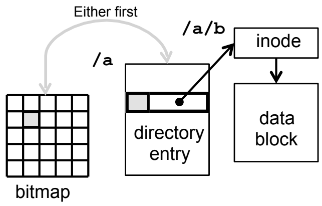
```
unlink:
    ...
    write directory block
      -- wait until done
    write bitmap block
      -- wait until done
    return
```

Figure 6.12: Synchronous disk writes for ext2 consistency.

[14]Actually it doesn't check, but rather "replays" all the changes recorded in the log.


Log:
[…]
[update entry in directory /a,
update bitmap to free block]
[…]

Step 1: record action in log



Step 2: write blocks in any order

Ext3 Journaling: The ext3 file system uses physical block logging: each log entry contains a header identifying the disk blocks which are modified (in the example you saw earlier, the bitmap and the directory entry) and a copy of the disk blocks themselves. After a crash the log is replayed by writing each block from the log to the location where it belongs. If a block is written multiple times in the log, it will get overwritten multiple times during replay, and after the last over-write it will have the correct value.

To avoid synchronous journal writes for every file operation, ext3 uses *batch commit*: journal writes are deferred, and multiple writes are combined into a single transaction. The log entries for the entire batch are written to the log in a single sequential write, called a *checkpoint*. In the event of a crash, any modifications since the last checkpoint will be lost, but since checkpoints are performed at least every few seconds, this typically isn't a problem. (If your program needs a guarantee that data is written to a file *right now*, you need to use the `fsync` system call to flush data to disk.)

Ext3 supports three different journaling modes:

- *Journaled*: In this mode, all changes (to file data, directories, inodes and bitmaps) are written to the log before any modifications are made to the main file system.
- *Ordered*: Here, data blocks are flushed to the main file system before a journal entry for any metadata changes (directories, free space bitmaps, inodes) is written to the log, after which the metadata changes may be made in the file system. This provides the same consistency guarantees as journaled mode, but is usually faster.
- *Writeback*: In this mode, metadata changes are always written to the log before being applied to the main file system, but data may be written at any time. It is faster than the other two modes, and will prevent the file system itself from becoming corrupted, but data within a file may be lost.

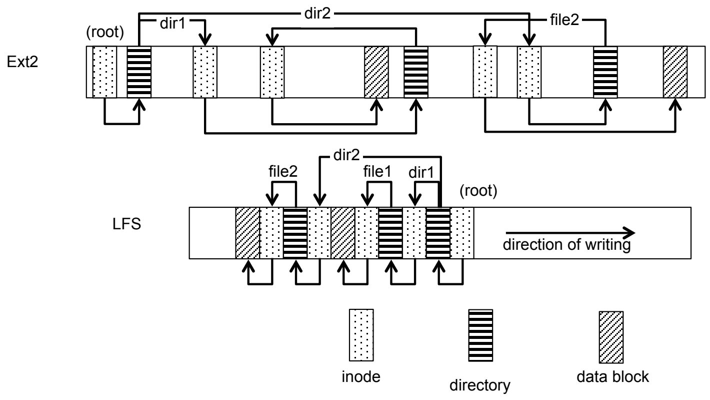


Figure 6.13: Ext2 vs Log-structured file system layout

6.9 Log-Structured File Systems

Log-structured file systems (like LFS in NetBSD, or NetApp WAFL) are an extreme version of a journaled file system: the journal is the entire file system. Data is never over-written; instead a form of copy-on-write is used: modified data is written sequentially to new locations in the log. This gives very high write speeds because all writes (even random ones) are written sequentially to the disk.

Figure 6.13 compares LFS to ext2, showing a simple file system with two directories (dir1, dir2) and two files (/dir1/file1, /dir2/file2). In ext2 the root directory inode is found in a fixed location, and its data blocks do not move after being allocated; in LFS both inode and data blocks move around—as they are modified, the new blocks get written to the head of the log rather than overwriting the old ones. The result can be seen graphically in the figure—in the LFS image, pointers only point to the left, pointing to data that is older than the block holding a pointer. Unlike ext2 there is no fixed location to find the root directory; this is solved by periodically storing its location in a small checkpoint record in a fixed location in the superblock. (This checkpoint is not shown in the figure, and would be the only arrow pointing to the right.)

When a data block is re-written, a new block with a new address is used. This means that the inode (or indirect block) pointing to the data block must be modified, which means that its address changes.

LFS uses a table mapping inodes to locations on disk, which is updated with the new inode address to complete the process; this table is itself

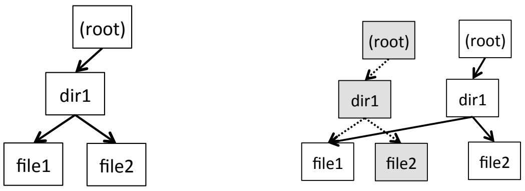


Figure 6.14: WAFL tree before and after update

stored as a file. (The astute reader may wonder why this update doesn't in fact trigger another update to the inode file, leading to an infinite loop. This is solved by buffering blocks in memory before they are written, so that multiple changes can be made.)

In WAFL these changes percolate all the way up through directory entries, directory inodes, etc., to the root of the file system, potentially causing a large number of writes for a small modification. (although they'll still be fairly fast since it's a single sequential write) To avoid this overhead, WAFL buffers a large number of changes before writing to disk; thus although any single write will modify the root directory, only a single modified copy of the root directory has to be written in each batch.

In Figure 6.14 a WAFL directory tree is shown before and after modifying /dir1/file2, with the out-of-date blocks shown in grey. If we keep a pointer to the old root node, then you can access a copy of the file system as it was at that point in time. When the disk fills up these out-of-date blocks are collected by a garbage collection process, and made available for new writes.

One of the advantages of a log-structured file system is the ability to easily keep snapshots of file system state—a pointer to an old version of the inode table or root directory will give you access to a copy of the file system at the point in time corresponding to that version. (e.g. look in your .snapshot directory on `login.ccs.neu.edu` - this data is stored on a NetApp filer using WAFL and its snapshot functionality.)

6.10 Kernel implementation

When applications access files they identify them by file and directory names, or by file descriptors (handles), and reads and writes may be performed in arbitrary lengths and alignments. These requests need to be translated into operations on the on-disk file system, where data is identified by its block number and all reads and writes must be in units of disk blocks.

The primary parts of this task are:

- Path translation - given a list of path components (e.g. "usr", "local", "bin", "program") perform the directory lookups necessary to find the file or directory named by that list.
- Read and write - translate operations on arbitrary offsets within a file to reads, writes, and allocations of complete disk blocks.

Path translation is a straightforward tree search - starting at the root directory, search for an entry for the first path component, find the location for that file or directory, and repeat until the last component of the list has been found, or an error has occurred. (not counting permissions, there are two possible errors here—either an entry of the path was not found, or a non-final component was found but was a file rather than a directory)

Reading requires finding the blocks which must be read, reading them in, and copying the requested data (which may not be all the data in the blocks, if the request does not start or end on a block boundary) to the appropriate locations in the user buffer.

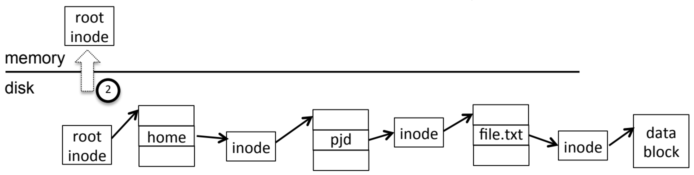
Writing is similar, with added complications: if a write starts in the middle of a block, that block must be read in, modified, and then written back so that existing data is not lost, and if a write extends beyond the end of the file new blocks must be allocated and added to the file.

As an example, to handle the system calls

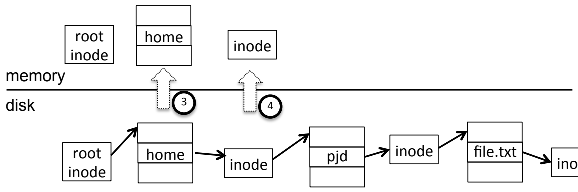
```
fd = open("/home/pjd/file.txt", O_RDONLY)
read(fd, buf, 1024)
```

the kernel has to perform the following steps:

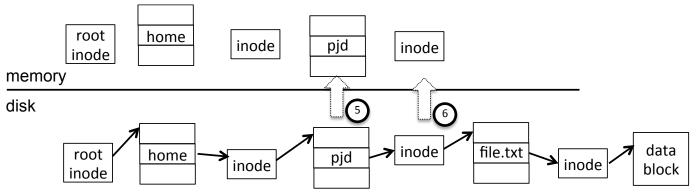
1. Split the string `/home/pjd/file.txt` into parts - `home`, `pjd`, `file.txt`
2. Read the root directory inode to find the location of the root directory data block. (let's assume it's a small directory, with one block)



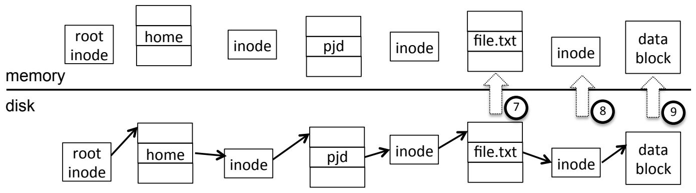
3. Read the root directory data block, search for `"home"`, and find the corresponding inode number
4. Read the inode for the directory `"home"` to get the data block pointer



5. Read the "`home`" directory data block, search for "`pjd`" to get the inode number
6. Read the "`pjd`" directory inode, get the data block pointer



7. Read the "`pjd`" directory block, and find the entry for `file.txt`
8. Read the "`file.txt`" inode and get the first data block pointer
9. Read the data block into the user buffer



Most of this work (steps 2 through 7) is path translation, or the process of traversing the directory tree to find the file itself. In doing this, the OS must handle the following possibilities:

1. The next entry in the path may not exist - the user may have typed `/hme/pjd/file.txt` or `/home/pjd/ffile.txt`
2. An intermediate entry in the path may be a file, rather than a directory - for instance `/home/pjd/file.txt/file.txt`
3. The user may not have permissions to access one of the entries in the path. On the CCIS systems, for instance, if a user other than pjd tries to access `/home/pjd/classes/file.txt`, the OS will notice that `/home/pjd/classes` is protected so that only user `pjd` may access it.

6.11 Caching

Disk accesses are slow, and multiple disk accesses are even slower. If every file operation required multiple disk accesses, your computer would run very slowly. Instead much of the information from the disk is cached in memory in various ways so that it can be used multiple times without going back to disk. Some of these ways are:

File descriptors: When an application opens a file the OS must translate the path to find the file's inode; the inode number and information from that inode can then be saved in a data structure associated with that open file (a *file descriptor* in Unix, or *file handle* in Windows), and freed when the file descriptor is closed.

Translation caching: An OS will typically maintain an in-memory translation cache (the dentry cache in Linux, holding individual directory entries) which holds frequently-used translations, such as root directory entries.

The directory entry cache differs from e.g. a CPU cache in that it holds both normal entries (e.g. directory+name to inode) and negative entries, indicating that directory+name does not exist[15]. If no entry is found the directory is searched, and the results added to the dentry cache.

Block caching: To accelerate reads of frequently-accessed blocks, rather than directly reading from the disk the OS can maintain a *block cache*. Before going to disk the OS checks to see whether a copy of the disk block is already present; if so the data can be copied directly, and if not it is read from disk and inserted into the cache before being returned. When data is modified it can be written to this cache and written back later to the disk.

Among other things, this allows small reads (smaller than a disk block) and small writes to be more efficient. The first small read will cause the block to be read into cache, while following reads from the same block will come from cache. Small writes will modify the same block in cache, and if a block is not flushed immediately to disk, it can be modified multiple times while only resulting in a single write.

Modern OSes like Linux use a combined buffer cache, where virtual memory pages and the file system cache come from the same pool. It's a bit complicated, and is not covered in this class.

[15]To be a bit formal about it, a CPU cache maps a *dense* address space, where every key has a value, while the translation cache maps a *sparse* address space.

6.12 VFS

In order to support multiple file systems such as Ext3, CD-ROMs, and others at the same time, Linux and other Unix variants use an interface called VFS, or the Virtual File System interface. (Windows uses a much different interface with the same purpose) The core of the OS does not know how to interpret individual file systems; instead it knows how to make requests across the VFS interface. Each file system registers an interface with VFS, and the methods in this interface implement the file system by talking to e.g. a disk or a network server.

VFS objects all exist in memory; any association between these structures and data on disk is the responsibility of the file system code. The important objects and methods in this interface are:

`superblock`. Not to be confused with the superblock on disk, this object corresponds to a mounted file system; in particular, the system *mount table* holds pointers to superblock structures. The important field in the superblock object is a pointer to the root directory `inode`.

`inode` - this corresponds to a file or directory. Its methods allow attributes (owner, timestamp, etc.) to be modified; in addition if the object corresponds to a directory, other methods allow creating, deleting, and renaming entries, as well as looking up a string to return a directory entry.

`dentry` - an object corresponding to a directory entry, as described earlier. It holds a name and a pointer to the corresponding `inode` object, and no interesting methods.

`file` - this corresponds to an open file. When it is created there is no associated "real" file; its open method is called with a `dentry` pointing to the file to open.

To open a file the OS will start with the root directory inode (from the superblock object) and call `lookup`, getting back a `dentry` with a pointer to the next directory, etc. When the dentry for the file is found, the OS will create a file object and pass the dentry to the file object's open method.

FUSE (File system in User Space) is a file system type in Linux which does not actually implement a file system itself, but instead forwards VFS requests to a user-space process, and then takes the responses from that process and passes them back to the kernel. This is seen in Figure 6.15, where a read call from the application results in kernel requests through VFS to FUSE, which are forwarded to a user-space file system process.

You will use FUSE to implement a file system in Homework 4, storing the file system in an image file accessed by the file system process.

- `getattr` - return attributes (size, owner, etc.) of a file or directory.
- `readdir` - list a directory
- `mkdir`, `rmdir`, `create`, `unlink` - create and remove directories and files
- `read`, `write` - note that these identify the offset in the file, as the kernel (not the file system) handles file positions.
- `rename` - change a name in a directory entry
- `truncate` - shorten a file
- ... and others, most of which are optional.

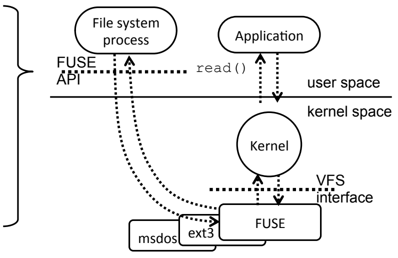


Figure 6.15: FUSE architecture and methods

Like VFS, the FUSE interface consists of a series of methods which you must implement, and if you implement them correctly and return consistent results, the kernel (and applications running on top of it) will see a file system. Unlike VFS, FUSE includes various levels of user-friendly support; we will use it in a mode where all objects are identified by human-readable path strings, rather than dentries and inodes.

6.13 Network File Systems

The file systems discussed so far are local file systems, where data is stored on local disk and is only directly accessible from the computer attached to that disk. Network file systems are used when we want to access to data from multiple machines - for instance, if you log in to a machine in the CCIS lab in room 102, your home directory will be the same on every machine, and is in fact stored on a NetApp file server in a machine room upstairs.

The two network file systems in most common use today are Unix NFS (Network File System) and Windows SMB (also known as CIFS). Each protocol provides operations somewhat similar to those in VFS (quite similar in the case of NFS, as the original VFS was designed for it), allowing the kernel to traverse and list directories, create and delete files, and read and write arbitrary offsets within a file.

The primary differences between the NFS (up through v3 - v4 is more complicated) and SMB are:

- State - NFS is designed to be stateless for reliability. Once you have obtained a file's unique ID (from the directory entry) you can just read from or write to a location in it, without opening the file. Operations are idempotent, which means they can be repeated multiple times without error. (e.g. writing page P to offset x can be repeated, while appending page P to the end of the file can't.) In contrast SMB is connection-oriented, and requires files and directories to be opened before they can be operated on. NFS tolerates server crashes and restarts more gracefully, but does not have some of the connection-related features in SMB such as authentication, described below.

- Identity - NFS acts like a local file system, trusting the client to authenticate users and pass numeric user IDs to the server. SMB handles authentication on the server side - each connection to the server begins with a handshake that authenticates to the server with a specific username, and all operations within that connection are done as that user.

Answers to Review Questions

6.1.1 False - otherwise there would be no subdirectories.

6.1.2 (2) and (4). If the file is deleted (2), all bytes (including the 1000th byte) will cease to exist; the 1000th byte is in the range being overwritten in (4). Renaming leaves the file otherwise unchanged, and modifying bytes 500 through 600 does not affect any other locations in the file.

6.1.3 (2) and (3). Bytes will be read starting at the current position until 'max' bytes have been read or the end of the file is reached, whichever comes first. The data itself is irrelevant, as is the 'buffer' argument. (as long as it points to enough valid memory)

6.1.1 (1) and (3). The OS will not allow non-empty directories to be deleted, as otherwise the files would be lost and their space would not be reclaimed. In addition it must prevent normal user writes to a directory, as user corruption of directory contents might be a security or crash risk.

6.1.2 (2) The mount table is internal to the kernel, and holds the current configuration of where filesystems are mounted, so it can be used when looking up a file. Programs external to the kernel are responsible for knowing where filesystems should be mounted, and doing so. (Typically the startup scripts in Linux read this information from the file `/etc/fstab`.)

6.2.1 True. Disks only support reading and writing in fixed-sized blocks; to modify a smaller region the OS must read a block, modify it, and write it back.

6.2.2 False. In almost all file systems, all blocks must be of the same size.

6.2.3 (1) Since the start of each file is indicated by only a block number (not by e.g. block number plus offset), each file must start at the beginning of a block.

6.2.4 False. It doesn't track free space at all, since being read-only it never needs to allocate space to create new files.

6.2.1 (3) Data blocks contain only data, and are linked via external pointers in the file allocation table.

6.2.2 (1) In ext2 multiple directory entries can point to the same inode. Like MS-DOS, ext2 requires (at least) one pointer per file block; these are in the inode and indirect blocks, while in the MS-DOS file system they are located in the FAT.