Filesystems & Their Performance

1 Introduction to File Systems

1.1 Introduction

- For most users, the file system is the most visible aspect of an operating system, apart from the user interface. Files store programs and data.
- The operating system implements the abstract `file' concept by managing I/O devices, for example, hard disks. Files are usually organised into directories to make them easier to use. Files also usually have some form of protections.
- The **file system** provides:
  - An interface to the file objects for both programmers and users.
  - The policies and mechanisms to store files and their attributes on permanent storage devices.

1.2 What's a File?

- A file is a *persistent* storage area for programs, source code, data, documents etc. that can be accessed by the user through her processes, and are long-term in existence (i.e exist after process-death, logouts and system crashes). There are many different file types, depending on the operating system and the application programs.

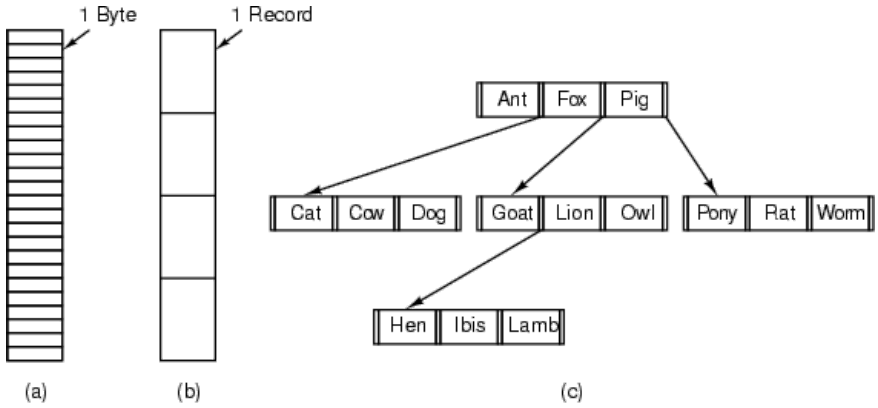


Figure 5-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

(from Tanenbaum)

- Examples are byte-structured, record-structures, tree-structured (ISAM). Most systems also give files *attributes*. Example attributes are:
  - File's name
  - Identification of owner
  - Size
  - Creation time
  - Last access time
  - Last modification time
  - File type

1.3 File Types

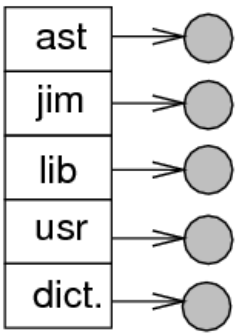
- Most operating systems have many different types. Under Unix, all files can be accessed as if they are byte-structured; any other structure is up to the application.
- In other operating systems, files are typed according to their content or their name, for example:
  - `file.pas`: Pascal source
  - `file.ftn`: Fortran source
  - `file.obj`: Compiler output
  - `file.exe`: Executable
  - `file.dat`: Data file
  - `file.txt`: Text file
- In some systems, the name is just a convention, and you can change the name of a file. In others, the conventions are enforced. You are not allowed to rename files, as the type would then change. This can be a hassle, for example, running `file.pas` through a Pascal source prettifier would produce `file.dat`, which could not be `converted' back to file.pas.

1.4 File Operations

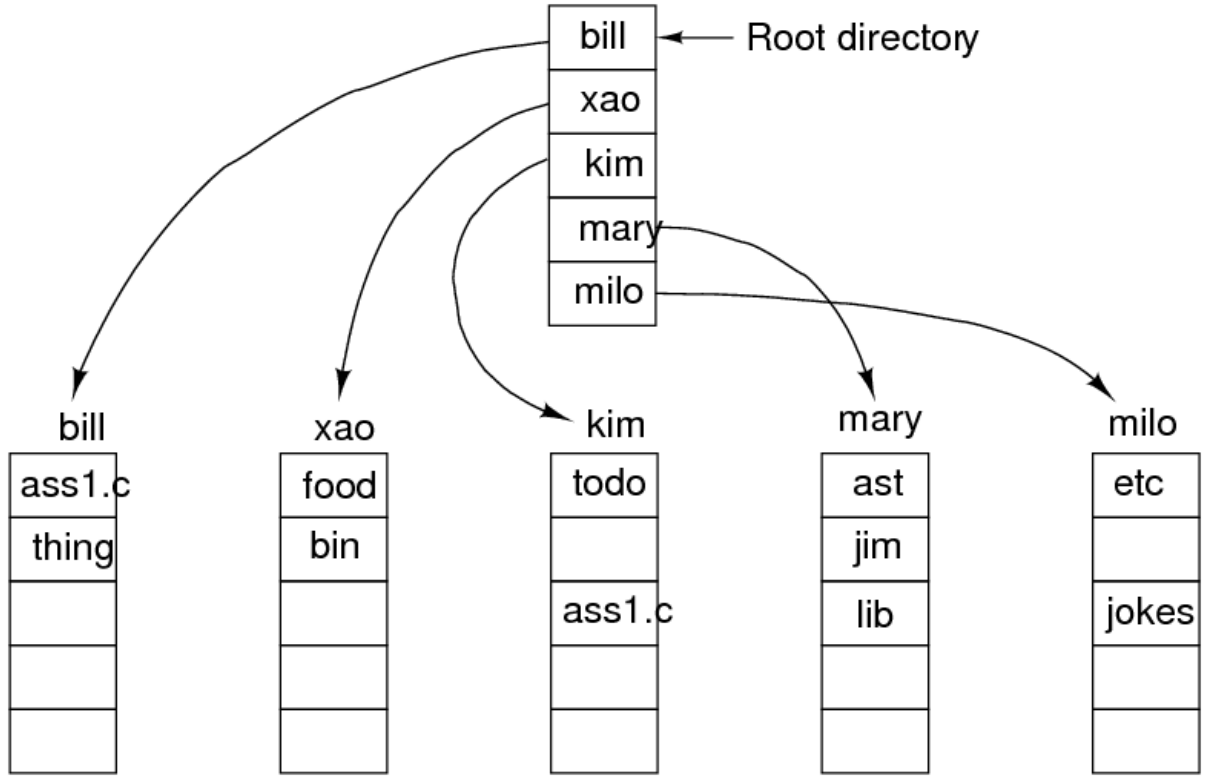
- Open, close, read, write (bytes or records) are always provided by every operating system.
- Random access is sometimes provided, and usually prevented on special files like terminals, where random access doesn't make sense etc. Record oriented files often have special operations to insert and delete records.

1.5 Directories

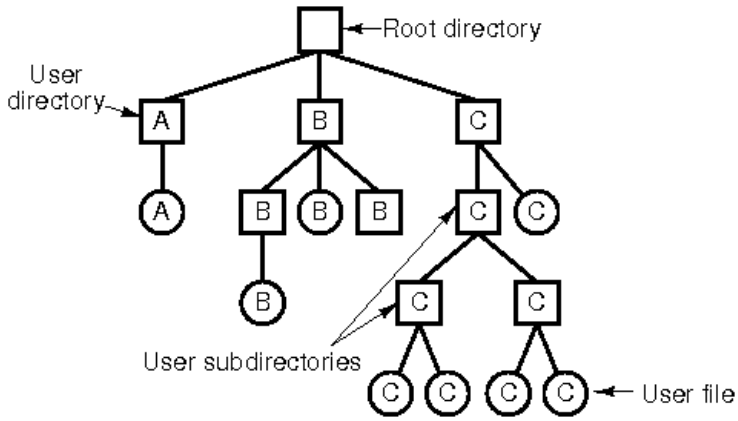
- Directories are used to help group files logically. They *only* exist to make life easier for humans. There are several types:
- Flat directory (single-level directory).



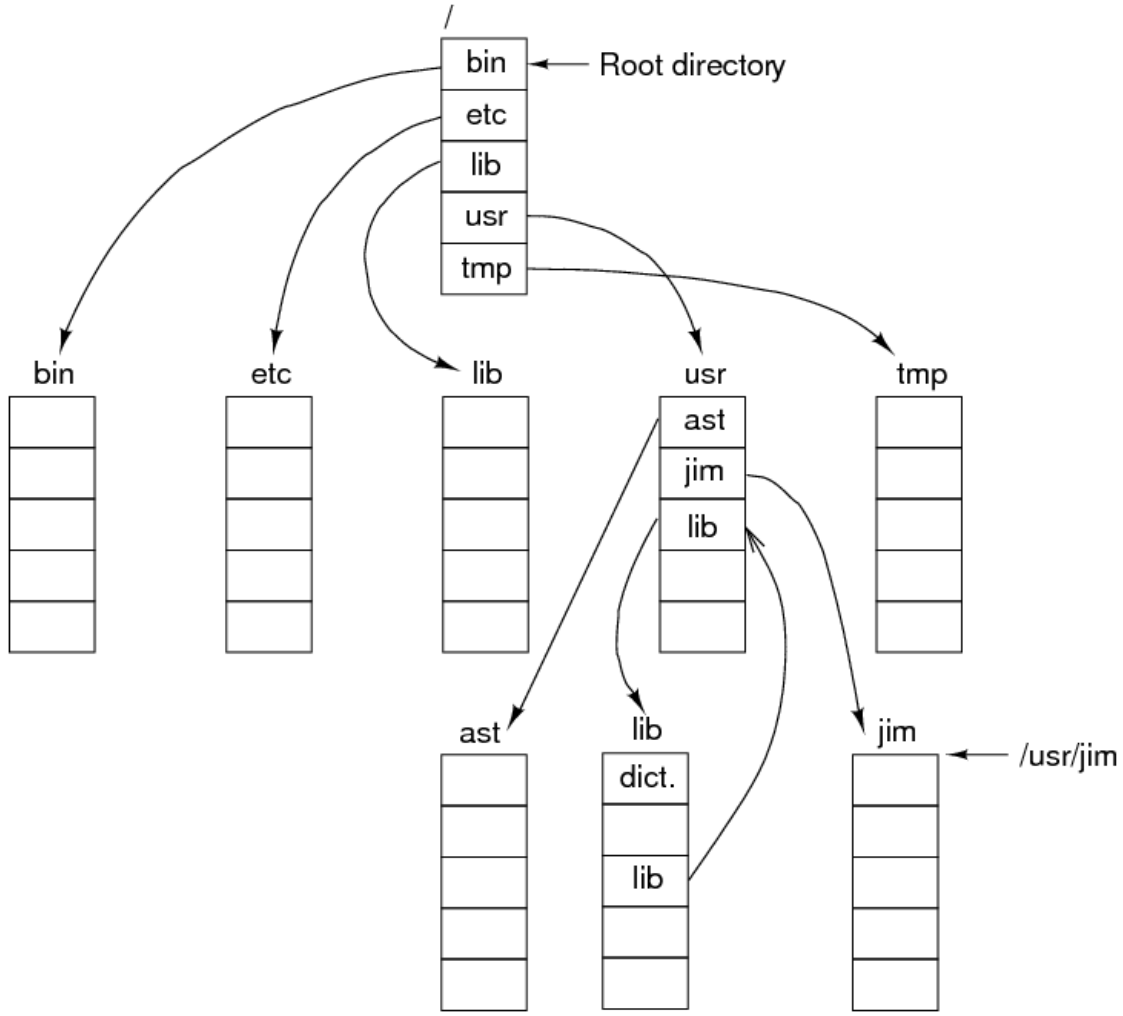
- Two-level directory (usually one directory per person).



- Tree-structured (hierachical).



- Finally, a general graph directory.



(all from Tanenbaum)

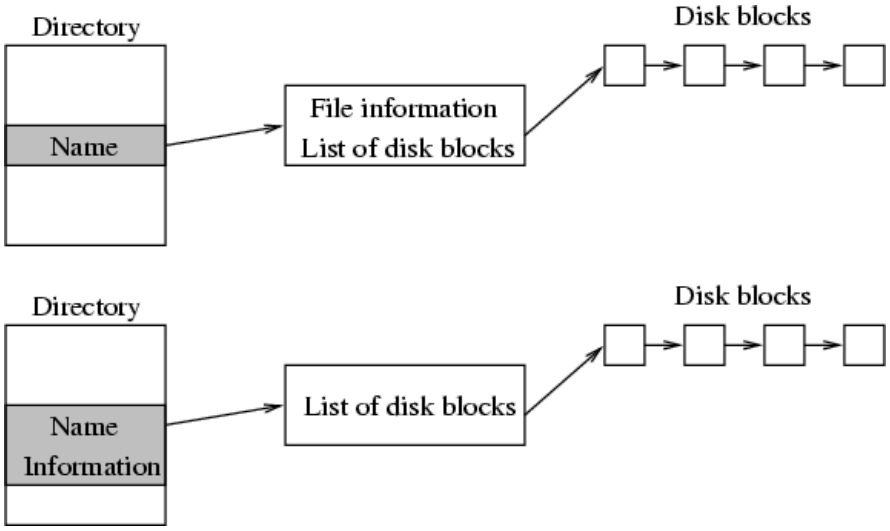
- The last directory structure can cause problems due to the file duplication. For example, a backup program may back the same file twice, wasting off–line storage.

1.6 Filesystem Metadata

- In order to know which blocks a file is composed of, and its attributes, a file system must also store *metatdata* for all files and directories. This metadata occupies some fraction of the disk space, typically 2% to 10%.
- Each file system stores different metadata, but examples are:
  - The name and attributes of each file.
  - The name and attributes of each directory.
  - The list of files and directories in each directory.
  - The list of blocks which constitute each file, in order.
  - The list of free blocks on the disk.
- We will consider the file system metadata in the next few sections.

1.7 Directory Information

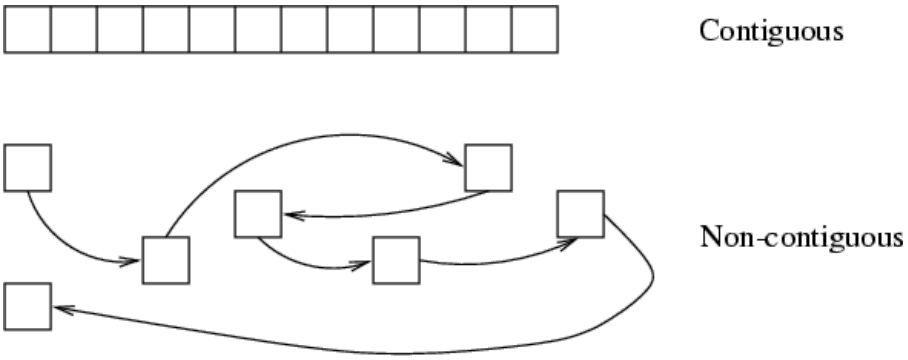
- The information about the directory structure (and the list of files in each directory) must be stored by the operating system on disk, so that it is maintained across shutdowns. There are two main methods for storing this information:



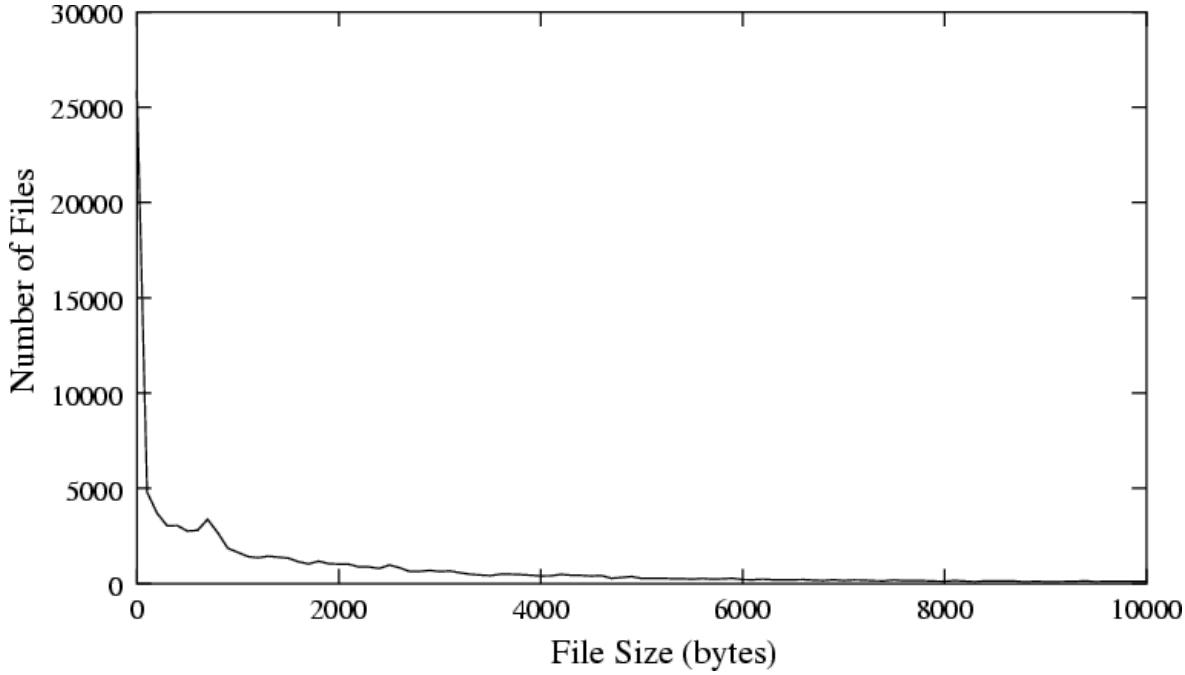
- The first method allows the acyclic and general directory structure, where a file may have multiple names and multiple attributes.

1.8 File System Design

- We now move on to the operating system point of view. The operating system needs to store arbitrarily–sized, often growing, sometimes shrinking, files on block–oriented devices. This can either be done in a **contiguous** fashion, or **non–contiguously**.



- A contiguous file layout is fast (less seek time), but it has the same problem as memory partitions: there is often not enough room to grow.
- A non–contiguous file layout is slower (we need to find the portion of the file required), and involves keeping a table of the file's portions, but does not suffer from the constraints that contiguous file layout does.
- The next problem is to choose a fixed size for the portions of a file: the size of the **blocks**. If the size is too small, we must do more I/O to read the same information, which can slow down the system. If the size is too large, then we get *internal fragmentation* in each block, thus wasting disk space. The average size of the system's files also affects this problem.



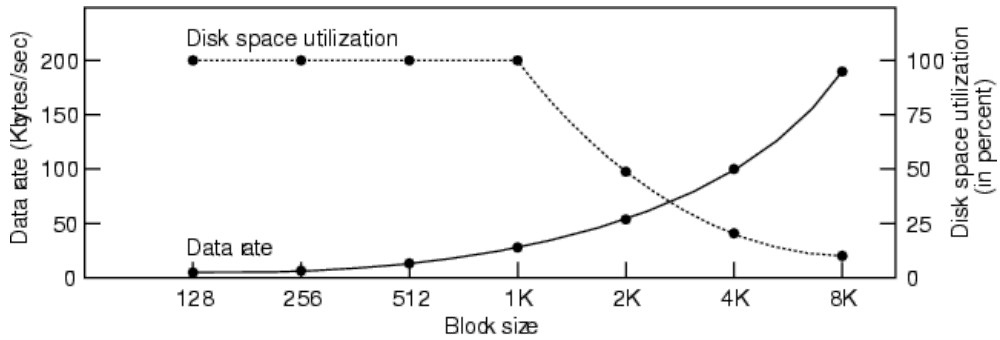
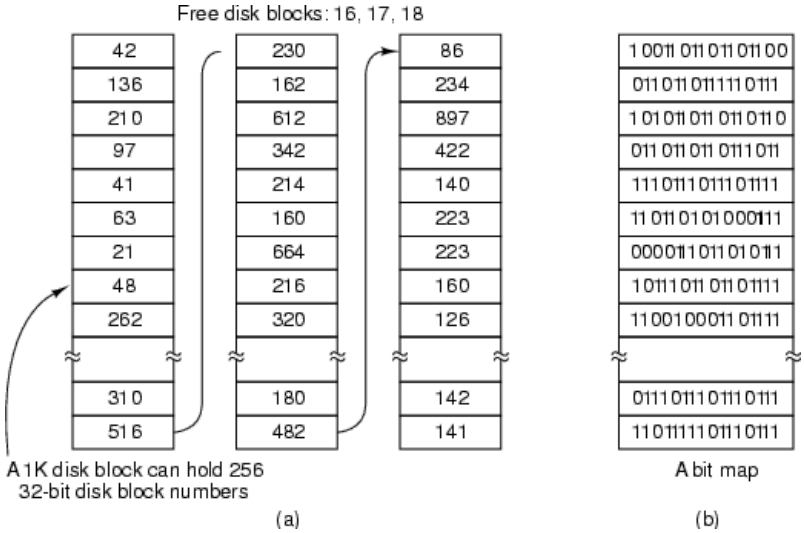


Figure 5-15. The solid curve (left-hand scale) gives the data rate of a disk. The dashed curve (right-hand scale) gives the disk space efficiency. All files are 1K.

(bottom figure from Tanenbaum)

- Because of these factors, a block size of 512 bytes up to 8K bytes is usually chosen.
- The operating system must know:
  - Which blocks on the disk are free (available for use).
  - For each file, which blocks it is using.
- The first is solved by keeping a list of free disk blocks. This list can often be large. For example, a 80G disk with a block size of 2K has 41,943,040 blocks. If each entry in the list is 4 bytes long, the free list is size 160M bytes. As the number of free blocks becomes smaller, the list size decreases.
- Another method of storing the list is with a **bitmap**. Have a table of n bits for a disk with n blocks. If a bit is 1, the corresponding block is *free*; otherwise, the block is in use. The bitmap table here is 1/32 the size of the free list above (i.e. only 5M bytes long), but is fixed in size.



(from Tanenbaum)

- Choosing a free block to use is easier in the list format: choose the first block on the list. Determining of block X is free is easier in the bitmap format: check to see if bit X is set to 1 or not.
- Usually the free list resides in physical memory, so that the operating system can quickly find free blocks to allocate. If the free list becomes too big, the operating system may keep a portion of the list in memory, and read in/out the other portions as needed. In any case, a copy of the free list **must** be stored on disk so that the list can be recovered if the machine is shut down.
- Thus, as well as the blocks holding file data, the operating system maintains *special* blocks reserved for holding free lists, directory structures etc. For example, a disk under Minix looks like:

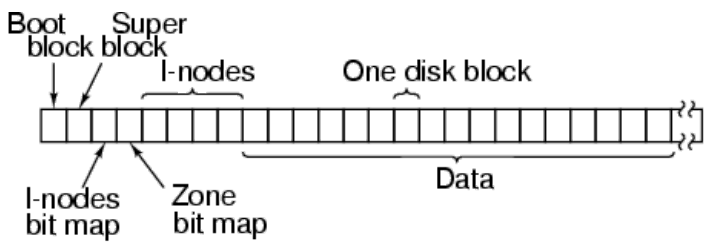


Figure 5-28. Disk layout for the simplest disk: a 360K floppy disk, with 128 i-nodes and a 1K block size (i.e., two consecutive 512-byte sectors are treated as a single block).

(from Tanenbaum)

- Ignore the i–nodes for now. The **boot** block holds the machine code to load the operating system from the disk when the machine is turned on.
- The **super** block describes the disk's geometry, and such things as:
  - the number of blocks on the disk
  - the number of i–node bitmap blocks
  - the number of blocks in the free bitmap
  - the first data block
  - the maximum file size
  - the first i–node bitmap block
  - the first block in the free bitmap
- The **i–nodes** are used to store the directory structure and the attributes of each file. More on these in future lectures.

- Note that disks sometimes suffer from physical defects, causing bad blocks. The free list can be used to prevent these bad blocks from being allocated. Mark a bad block as being used; this will prevent it from being allocated in the future.
- An alternative is to use a special value to indicate that the block is bad; Windows FAT filesystems use this technique, for example. Bad blocks cannot be marked with a free bitmap, because there are only two values per block: free and in–use.

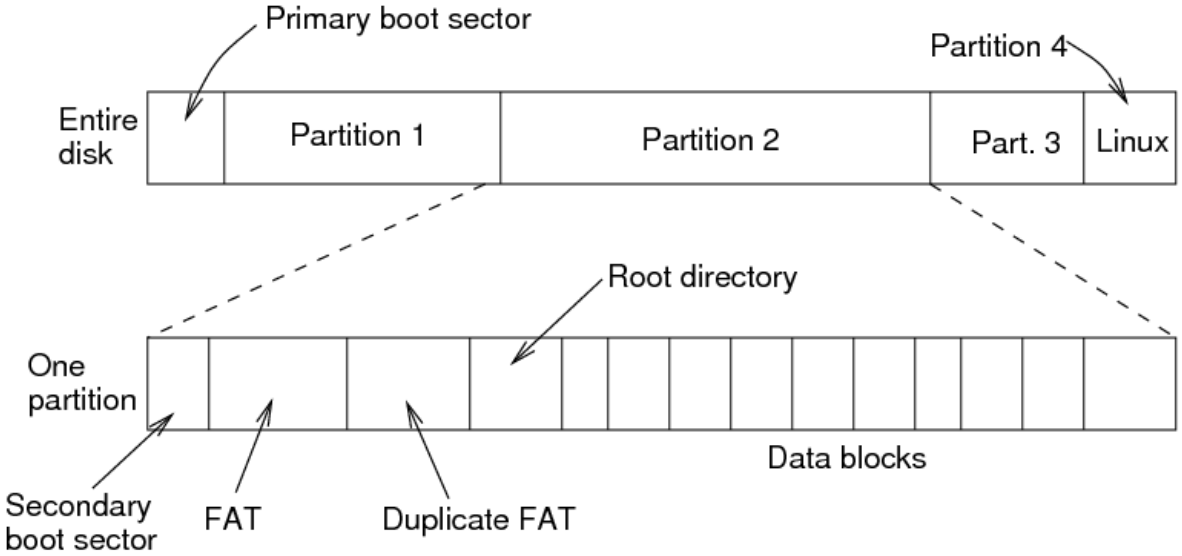
2 File System Layout

2.1 Introduction

- The data and attributes of files must be stored on disk to ensure their long–term storage. Different file systems use different layouts of file data and attributes on disk. We will examine the filesystems for Windows and Unix.

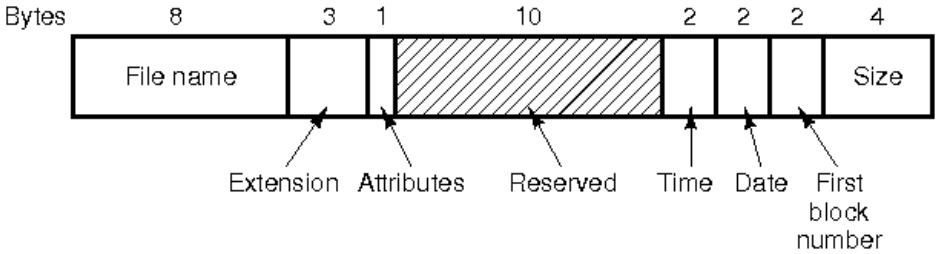
2.2 The FAT Filesystem

- Windows breaks disks into up to four sections, known as **partitions**. The first block holds the *primary boot sector*, which describes the types and sizes of each partition.



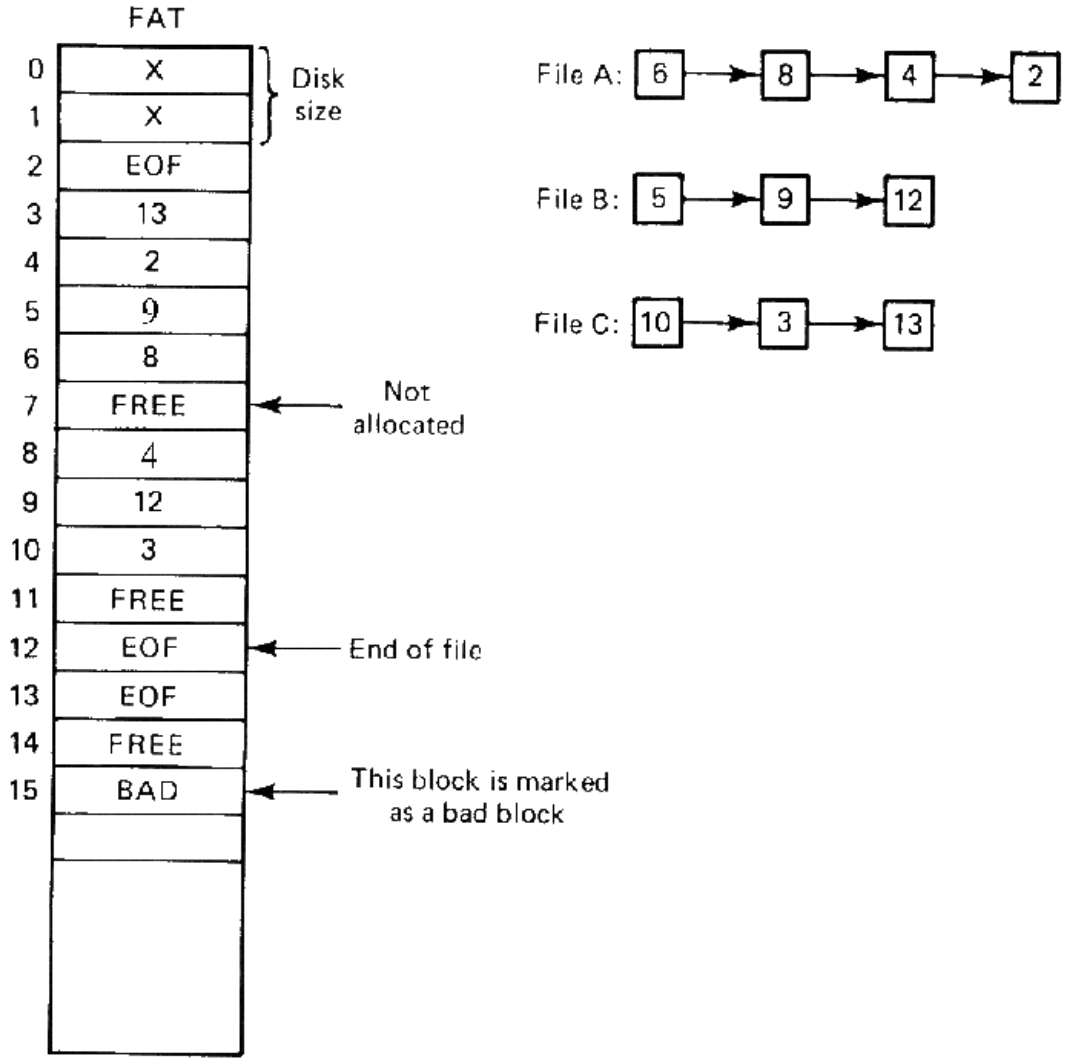
(from Tanenbaum)

- Recent versions of Windows use the NTFS filesystem in each partition; earlier versions used the FAT filesystem.
- FAT is also used on most USB drives and flash cards in cameras, media players etc., which is why we can still consider it.
- On a FAT partition there is a secondary boot block, a **file allocation table** or **FAT**, a duplicate FAT, a root directory and a number of blocks used for file storage.
- Each file has 32 bytes of attributes which are stored in each directory as a *directory entry*. The entry describes the first block used for data storage.



(from Tanenbaum)

- The list of blocks used by the file are kept in the FAT. The FAT lists all disk blocks, and describes if the block is free or bad, or which block *comes next* in the file. The last block in a file is marked with a special EOF value to indicate this. Thus, the FAT is really a big set of linked lists, with each block pointing to the next one, for every file.



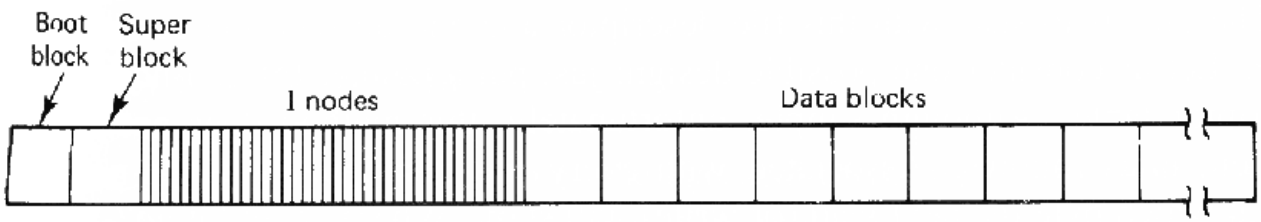
(from Tanenbaum)

- System that use FAT will keep some of the FAT in memory to speed lookups through the table. For large disks, the FAT may be large, and must be stored in several disk blocks.
- The size (in bits) of a FAT entry limits the size of the filesystem. For example, MS–DOS originally used a 320K floppy with 512–byte blocks numbered 0 – 639, using 12 bits to number each block. Thus, the largest FAT (640 block numbers, 12 bits per number) requires only 960 bytes, which fits into two disk block.
- When hard disks with more than 2^{12}=4096 blocks arrived, the 12–bit block numbers were too small. So MS–DOS had to move to 16–bits, causing the FAT structure and the directory structures to change.
- FAT–16 provides for 2^{16}=65,636 block numbers. Assuming that a block is 512–bytes, then disks are limited to 65,636 blocks, or 32M bytes in size. When disks bigger than 32M arrived, the FAT–16 structure was changed to deal with **clusters** of blocks. MS–DOS and Windows 95 could only read/write in clusters. If you had a 10G disk drive, your clusters were 160K in size. If you stored a 20–line text file, it would occupy 160K on the disk.
- The size of the FAT depends on the number of clusters. With FAT–16, if there are 65,536 clusters, and each one has a 16–bit cluster number, then the FAT occupies 128M bytes of disk.

- FAT-32 was introduced to fix the cluster problem. Block numbers are 32-bits in size, and the FAT only has to be big enough to list every block. For a 120G disk drive with 1K blocks, there are 120 million 4-byte block numbers, or 480M bytes for the FAT.
- One drawback of the FAT is that it affects the mechanism to search through the FAT to find a file's list of blocks. This is fine for sequential access, but penalises any random access in the file.
- Windows performs *first free* block allocation. This may lay a file's data blocks out poorly across the disk. **Defragmentation** can be performed to make files contiguous.
- Loss of the FAT makes a filesystem unusable. This is why there is a duplicate FAT on each FAT partition. Viruses, however, usually destroy both FATs.

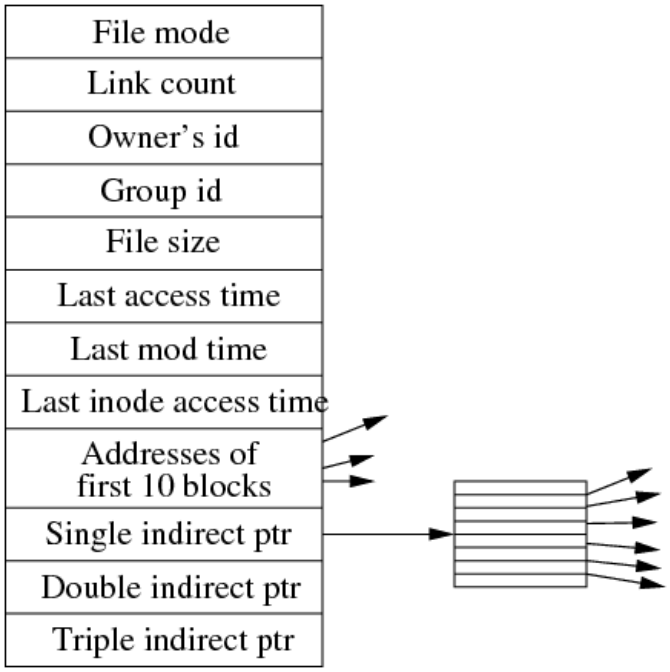
2.3 The System V Unix Filesystem

- The System V filesystem is shown below. The first block is reserved as a boot block. The second block is the **superblock**. It contains information about the layout of the filesystem, such as the number of i-nodes, the number of blocks on the disk, and the start of the list of free disk blocks.



(from Tanenbaum)

- After the superblock are the **i-nodes**. These contain the attributes of files, but not the filenames. Note that the attributes are quite different to the FAT filesystem. As Unix is a multiuser system, files have ownership and a three-level set of protections (read, write, execute for user, a group of users, and all other users),



(from Tanenbaum)

- Instead of a linear structure for keeping the list of blocks, Unix uses a tree structure which is faster to traverse.
- The i-node holds the first 10 block numbers used by the file. If the file grows larger, a disk block is used to store further block numbers; this is a **single indirect block**. Assume the single indirect block can hold 256

block numbers: this allows the file to grow to 10+256=266 blocks.

- If the single indirect block becomes full, another *two* blocks are used. One becomes the next single indirect block, and the second points to the new single indirect block; this is a **double indirect block**, which can point at 256 single indirect blocks, allowing the file to grow to 10+256+256*256=65,802 blocks.
- Sometimes, a file will exceed 65,802 blocks. In this situation, a **triple indirect block** is allocated which points at up to 256 double indirect blocks. There can only be one triple indirect block, but when used, a file can grow to be $10+256+256^2+256^3$=16,843,018 blocks, or around 16 gigabytes in size.
- One strength of the i–node system is that the indirect blocks are used only as required, and for files less than 10 blocks, none are required. The main advantage is that a tree search can be used to find the block number for any block, and as the tree is never more than three levels deep, it never takes more than three disk accesses to find a block. This also speeds random file accesses.
- A System V Unix directory entry looks like:

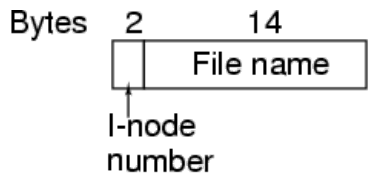


Figure 5-13. A UNIX directory entry.

(from Tanenbaum)

- Note here that Unix only stores the file name and i–node number in each directory entry. The rest of the information about each file is kept in the i–node. This allows files to be **linked**, allowing acyclic and general graph directory structures. This cannot be done with the FAT filesystem.

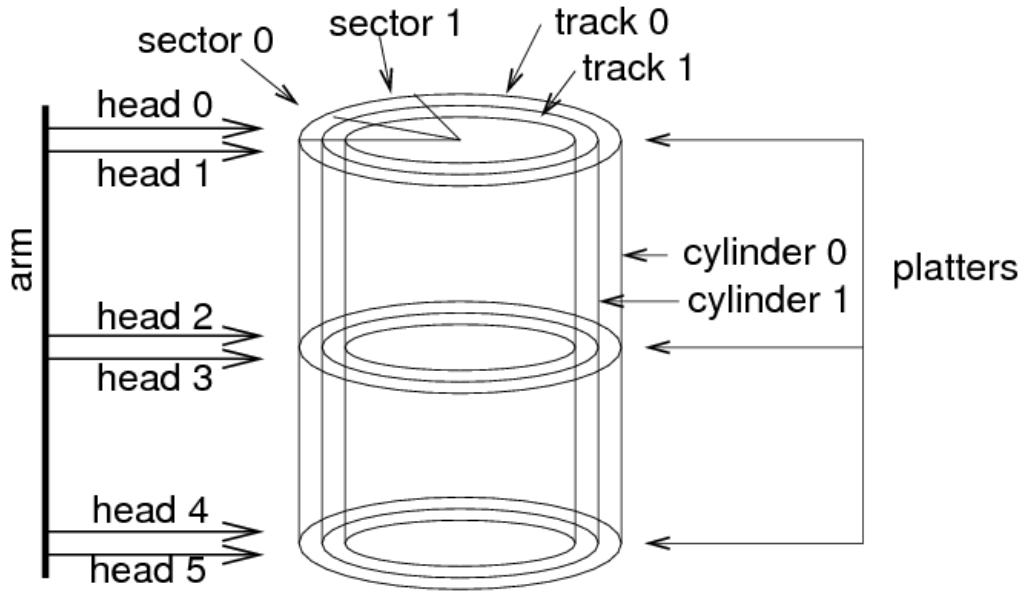
592	shared.c in /usr/fred

592	temp_shared.c in /tmp

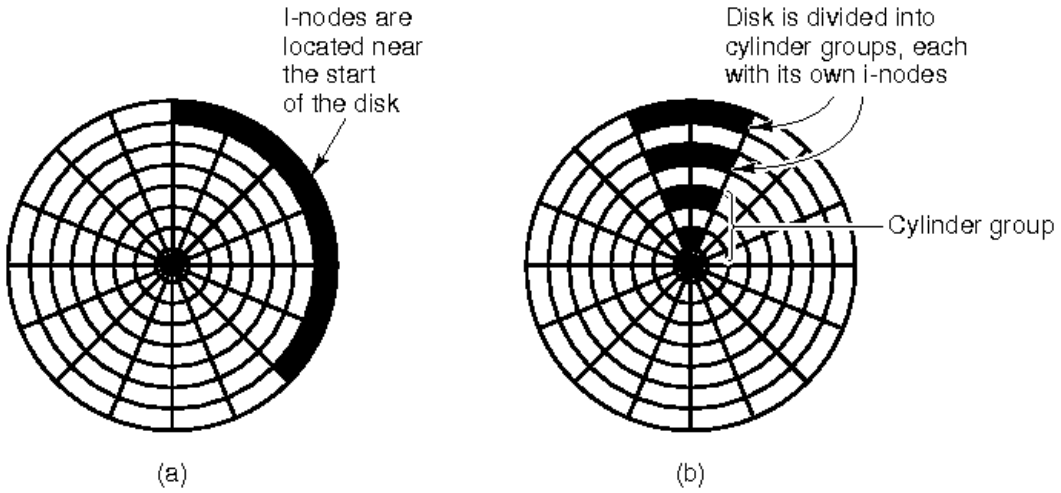
- One problem, if the file `shared.c` is deleted, will the other file still exist? To prevent this, Unix keeps a **link count** in every i–node, which is usually 1. When a file is removed, its directory entry is removed, and the i–node link count decremented; if the link count becomes 0, the i–node and the data blocks are removed.
- Linked files have been found to be useful, but there are problems with links. Programs that traverse the directory structure (e.g backup programs) will backup the file multiple times. Even worse, if a directory is a link back to a higher directory (thus making a loop), files in between may be backed up an infinite number of times. Thus, tools must be made clever enough to recognise and keep count of linked files to ensure that the same file isn't encountered twice.
- The System V filesystem has several problems. The performance ones are discussed below. The list of i–nodes is fixed, and thus the number of files in each filesystem is fixed, even if there is ample disk space. Unix uses a bitmap to hold the list of free disk blocks, and so bad blocks are not easily catered for.

2.4 The Berkeley Fast Filesystem

- The System V filesystem did not provide good performance. The original block size of 512 bytes gave slow I/O, and the block size was increased to 1,024 bytes. Having all i–nodes at the beginning of the disk caused large head movement as a file's data and attributes were at different ends of the disk. Files that grew slowly ended up with non–contiguous allocation, and the 14 character filename was restrictive.
- A new filesystem called the Berkeley Fast Filesystem (FFS) was developed at the University of California, Berkeley, to address these problems. FFS took into consideration the fact that head movement was expensive, but a **cylinder's** worth of data could be accessed without any head movement.



- FFS breaks the filesystem up into several *cylinder groups*. Cylinder groups are usually a megabyte or so in size. Each cylinder group has its own free block list and list of i–nodes.



(from Tanenbaum)

- The superblock is replicated in each cylinder group, to minimise the problem of a corrupt filesystem due to loss of the superblock.
- To improve I/O speeds, blocks come in two sizes, with the smaller size called a *fragment*. Typical block/fragment sizes are 8K/1K. An FFS file is composed entirely of blocks, except for the last block which may contain one or more consecutive fragments.
- Unused fragments may be used by other files, and occasional recopying must be done as files grow in size, to either merge fragments into single blocks, or to keep fragments consecutive within a block.
- The Standard I/O library uses the block size to perform file I/O, which helps to maintain I/O performance.
- FFS has several allocation policies to improve performance:
  - Inodes of files within the same directory are placed in the same cylinder group.
  - New directories are in different cylinder groups than their parents, and in the one with the highest free i–node count.
  - Data blocks are placed in the same cylinder group as the file's i–node.
  - Avoid filling a cylinder group with large files by changing to a new cylinder group at the first indirect block, and at one megabyte thereafter.
  - Allocate sequential blocks at rotationally optimal positions.

- FFS also increased the filename to 255 characters maximum. FFS provides a significant increase in I/O performance against the System V filesystem (typically several hundred percent improvement).
- The Linux ext3fs filesystem was heavily influenced by the Berkeley FFS. ext3fs also distributes i–nodes across the disk in groups, and tries to keep file blocks close to the corresponding i–nodes. As with FFS, this helps to minimise arm motion.

3 File System Reliability & Performance

3.1 File System Reliability

- The file system is the repository of user information on a computer system. If a CPU or memory is destroyed, another can be bought. But if a disk is damaged, or the files on the disk damaged, the information therein cannot be restored. Note that even if the data is intact, but the FATs, inodes or directory structure is damaged, the data in the files is effectively lost.
- For most users, the consequence of data loss is catastrophic. The operating system must provide methods and tools to minimise the possibility of loss, and also minimise the effect if any loss occurs.

3.2 Bad Blocks

- Disks come with bad blocks, and they appear as the disk is used. Each time a bad block appears, mark that block as **used** in the free block table, but ensure that no file uses the block. This leaves the problem of fixing any file that was using the bad block.

3.3 Backups

- Generally, the best method of ensuring minimal effect after a catastrophic loss is to have a copy of the data on another medium (disk/tapes); this is a **backup**. It is impossible to have a backup completely up to date, therefore even with a backup you may still lose some data.
- Backups should be done at regular intervals, and the *entire* contents of the file system is transferred to disk/tape – a **full** backup. Usually weekly or monthly.
- If the file system is large (e.g greater than 10G), backups are very big and time consuming. Between full backups, do **incremental** backups i.e. copy only those files that have changed since the last full/incremental backup.
- It is also a good idea to keep 3 **full** dumps, and rotate them when doing a full backup. Thus, even when you are doing a full backup and writing over the physical medium, you still have the last two full backups intact.

3.4 File System Consistency

- It is important to keep the file system consistent. If the system crashes before all modified blocks have been written out to disk, the file system is inconsistent, and when the system is rebooted, this may produce weird or unpleasant effects; this is especially true if the file system structures (FATs, inodes, directories, free list) are inconsistent.
- Most operating systems have a program that checks the file system consistency. This is usually run when the system reboots. It may do the following:
- Check that the link count in an inode equals the number of files pointing to that inode.
- Check that all file/directory entries are pointing at used inodes.
- Block allocation consistency:
    1. Read in the free list/bitmap
    2. Build a **used** list by going through all the inodes/FATs and marking the blocks that are used.
    3. If the file system is consistent, then each block will have a `1' on the free list or a `1' on the used list.

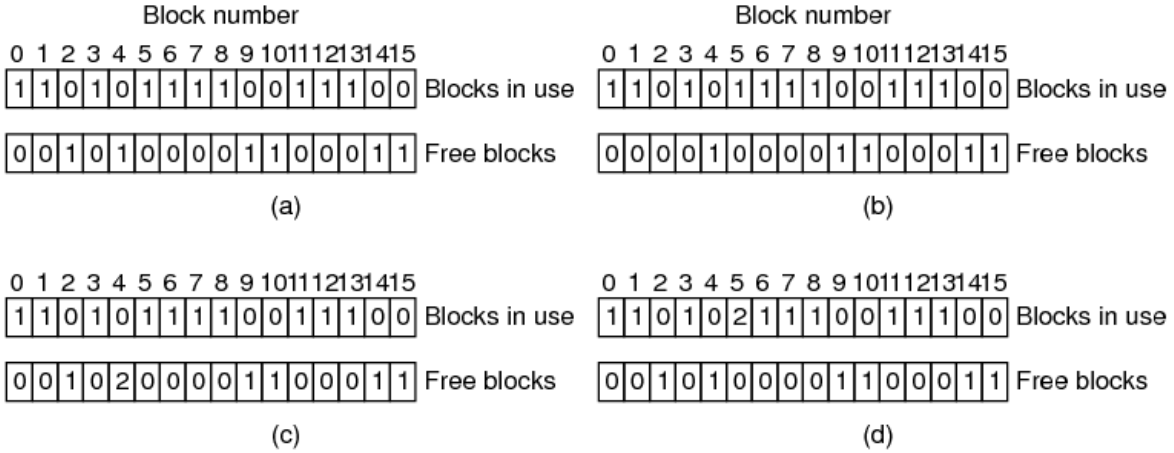


Figure 5–18. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

(from Tanenbaum)

4. Missing block # 2. This could be a **bad** block, or a free block that hasn't been marked.
5. Duplicate free block #4. This can be ignored.
6. Duplicate **used** block #5, i.e 2 or more files use this block. This should never happen! If one file is freed, the block is on both the used and the free list! One solution is to make `n' copies of the block and give each file its own block. Also signal an error to the owners of each file.
7. Block on both free and used lists. We must fix this immediately, as the free block may be allocated and overwritten. Remove the block from the free list.

- Some operating systems can protect against user errors. For example, under Windows, you can usually undelete a file that was accidentally deleted, because the directory entry is marked unused, but the remaining information is still intact (i.e attributes, FAT tables). This is technically possible under Unix, but usually not possible because Unix allows multiple processes and users, thus the freed blocks may be reused at any time.

3.5 File System Performance – Caching

- Disk accesses are much slower than memory, usually at least 10,000 times slower. We therefore need to optimise the file system performance. We have seen that arm and sector scheduling can help in this regard.
- Another common technique is a **cache**: a collection of disk blocks that logically belong on disk, but are kept in **memory** to improve performance.
- Obviously, we should keep the most recently used or most frequently used disk blocks in memory so that read/writes to these blocks will occur at memory speeds.
- Note that we can use the cache to cache **reads** from the disk *and* **writes** to disk. This is known as a **write–through cache**, as writes go through the cache before eventually being written to the disk.
- Some operating systems only cache reads, and all writes go directly to disk. This is known as a **write–back cache**. The problem arises as to which block to discard/write back to disk if the cache is full? This is very much like out **paging** problem, and algorithms like FIFO, second chance and LRU can be used here. Note that because block accesses occur much less frequently than page accesses, it is feasible to keep cached blocked in strict LRU order.
- Any algorithm that is used should take into account the fact that blocks containing inodes, directories and free lists are special because they are essential to the file system consistency, and are likely to be used frequently. For this reason, some operating systems will write metadata to disk immediately, but writes file data *through* the cache.
- One problem with writes *through* the cache is that, if the machine crashes, **dirty** blocks will not be written to the disk, and so data will be lost. This means recent disk writes are not written to disk.

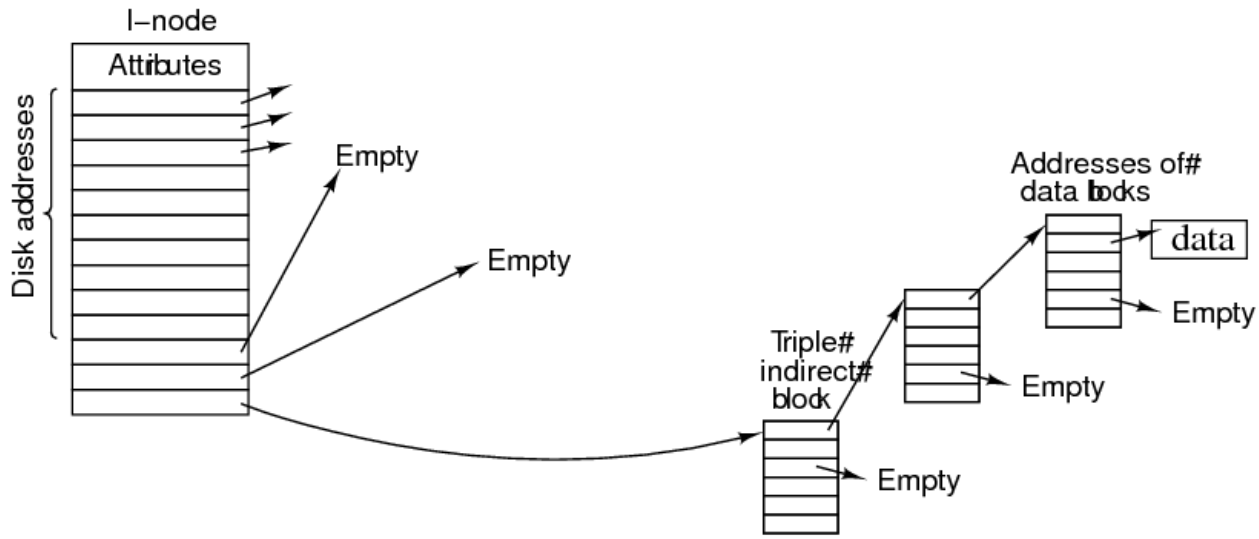
- The solution under Unix is to **flush** all dirty blocks to disk every 30 seconds. Caches that write **direct** to disk do not suffer from this problem, but are usually slower because of the I/O delay in writing.
- The larger the cache, the more blocks in memory, and thus the better the **hit-rate** and the performance. Disk caches are usually a small fraction of the computer's RAM, e.g. 16M bytes or so.
- Finally, it's a bit quirky that disks are used to hold pages from virtual memory, and memory is used to hold recently used disk blocks. It makes sense when you realise that VM is providing more memory with a performance penalty, whereas disk caching is providing some disk with a performance increase.
- In fact, some operating systems (Solaris, FreeBSD, Linux) have merged the disk cache into the virtual memory subsystem. Disk blocks and memory pages are kept on the same LRU list, and so the whole of a computer's memory can be used for both working set storage and disk caching.

3.6 File Block Allocation

- Remember that disk arm motion is slow compared with rotational delay. Therefore the operating system should attempt to allocate blocks to a file contiguously (or with minimal arm movement) where possible. This will lower requested arm motion, and improve disk access.
- Some operating systems (notably those for micros) have utilities that take a file system, and rearrange the used blocks to that each file has its blocks arranged in contiguous order. This is really only useful for those files which will not grow. Files that do grow will have some blocks on one area of disk and others on another disk area.

3.7 Holey Files

- No, we're not talking about religious files here. Imagine opening a new file, seeking to character 1,000,000 and writing `Hello' into the file. Technically, the file is 1,000,005 characters long, but it is mostly empty space. A file system that supports **holey** files only allocates blocks that have something in them. For example, in Unix, we would only have the following blocks allocated:



- Other data/indirect blocks are allocated only as needed. Reads on unallocated blocks will return `empty' blocks i.e. all zeroes.

File translated from TEX by TTH, version 3.85.
On 8 Dec 2010, 09:06.