Chapter 11 File System
(Revision number 10)

In this chapter, we will discuss issues related to mass storage systems. In particular, we will discuss the choices in designing a file system and its implementation on the disk (hard drive as it is popularly referred to in the personal computer parlance). Let us understand the role of the file system in the overall mission of this textbook in "unraveling the box". Appreciating the capabilities of the computer system is inextricably tied to getting a grip on the way information is stored and manipulated inside the box. Therefore, getting our teeth into how the file system works is an important leg of the journey into unraveling the box.

We have all seen physical filing cabinets and manila file folders with papers in them. Typically, tags on the folders identify the content for easy retrieval (see Figure 11.1). Usually, we may keep a directory folder that tells us the organization of the files in the filing cabinet.

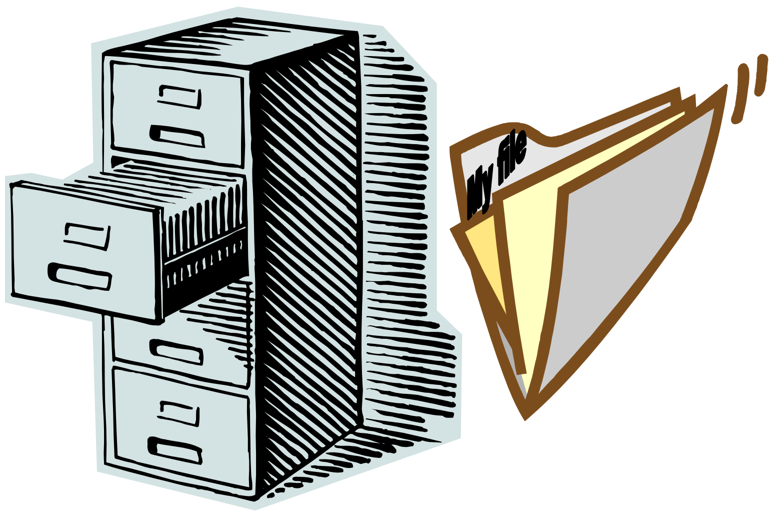


Figure 11.1: File cabinet and file folder

A computer file system is similar to a physical filing cabinet. Each file (similar to a manila file folder) is a collection of information with attributes associated with the information. Process is the software abstraction for the processor; data structure is the software abstraction for memory. Similarly, a file is the software abstraction for an input/output device since a device serves as either a source or sink of information. This abstraction allows a user program to interact with I/O in a device independent manner.

First, we will discuss the attributes associated with a file and the design choices therein, and then consider the design choices in its implementation on a mass storage device.

11.1 Attributes

We refer to the attributes associated with a file as *metadata*. The metadata represents *space overhead* and therefore requires careful analysis as to its utility.

Let us understand the attributes we may want to associate with a file.

- **Name:** This attribute allows logically identifying the contents of the file. For example, if we are storing music files we may want to give a *unique name* to each recording. To enable easy lookup, we may keep a *single* directory file that contains the names of all the music recordings. One can easily see that such a *single level* naming scheme used in early storage systems such as Univac Exec 8 computer systems (in the 70's), is restrictive. Later systems such as DEC TOPS-10 (in the early 80's) used a two-level naming scheme. A top-level directory allows getting to an individual user or a project (e.g. recordings of Billy Joel). The second level identifies a specific file for that user/project (e.g. a specific song).

  However, as systems grew in size it became evident that we need a more hierarchical structure to a name (e.g. each user may want to have his/her own music collection of different artists). In other words, we may need a multi-level directory as shown in Figure 11.2.

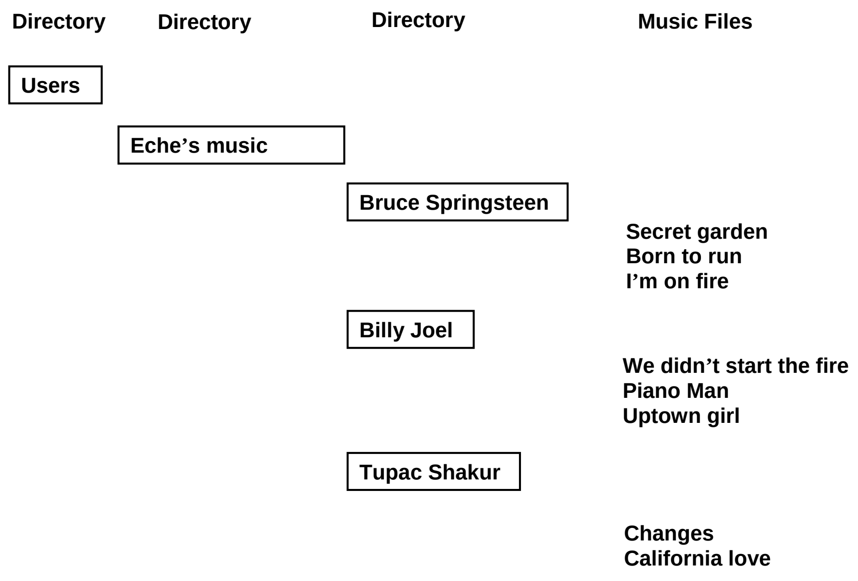


Figure 11.2: A multi-level directory

  Most modern operating systems such as Windows XP, Unix, and MacOS implement a multi-part hierarchical name. Each part of the name is *unique* only with respect to the previous parts of the name. This gives a tree structure to the organization of the files in a file system as shown in Figure 11.3. Each node in the tree is a name that is unique with respect to its parent node. Directories are files as well. In the tree structure shown in Figure 11.3, the intermediate nodes are *directory files*, and the leaf nodes are *data files*. The contents of a directory file are information about the files in the next level of the subtree rooted at that directory file (e.g., contents of directory

`users` are {**`students, staff, faculty`**}, contents of directory **`faculty`** are member of the faculty such as **`rama`**).

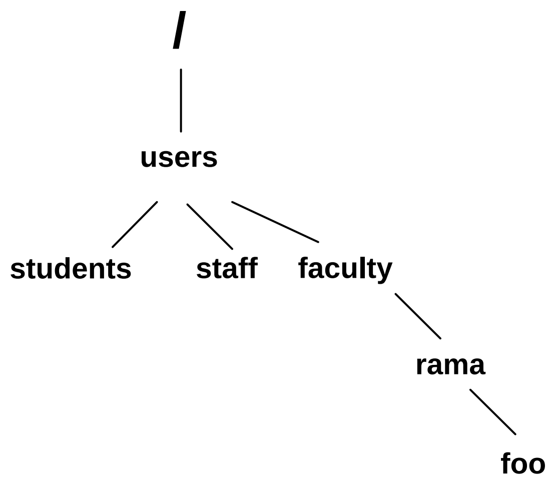


Figure 11.3: A tree structure for the file name /users/faculty/rama/foo

Some operating systems make extensions (in the form of suffixes) to a file name mandatory. For e.g. in DEC TOPS-10 operating system, text files automatically get the .TXT suffix appended to the user supplied name. In UNIX and Windows operating systems, such file name extensions are optional. The system uses the suffix to guess the contents of the file and launch the appropriate application program to manipulate the file (such as the C compiler, document editor, and photo editor).

Some operating systems allow *aliases* to a file. Such aliases may be at the level of the actual content of the file. Alternatively, the alias may simply be at the level of the names and not the actual contents. For example, the **`ln`** command (stands for link) in UNIX creates an alias to an existing file. The command

```
ln foo bar
```

results in creating an alias **`bar`** to access the contents of an existing file named **`foo`**. Such an alias, referred to as a *hard link*, gives an equal status to the new name **`bar`** as the original name **`foo`**. Even if we delete the file **`foo`**, the contents of the file are still accessible through the name **`bar`**.

i-node access rights hard links size creation time name

```
3193357 -rw-------   2 rama 80 Jan 23 18:30 bar
3193357 -rw-------   2 rama 80 Jan 23 18:30 foo
```

We will explain what an i-node is shortly in Section 11.3.1. For now, suffice it to say that it is a data structure for the representation of a file. Note that both **`foo`** and **`bar`** share exactly the same internal representation since they both have the same i-node. This is how both the names have equal status regardless of the order of creation of

foo and **bar**. This is also the reason why both the names have the size, and the same timestamp despite that fact that the name **bar** was created later than **foo**. Contrast the above situation with the command

```
ln –s foo bar
```

This command also results in creating an alias named **bar** for **foo**.

```
i-node   access rights   hard links      size   creation time    name

3193495 lrwxrwxrwx   1 rama  3 Jan 23 18:52 bar -> foo
3193357 -rw-------   1 rama 80 Jan 23 18:30 foo
```

However, the difference is in the fact that **bar** is *name equivalence* to **foo** and does not directly point to the file contents. Note that the i-nodes for the two names are different. Thus, the time of creation of **bar** reflects the time when the **ln** command was actually executed to create an alias. Also, the size of the files are different. The size of **foo** depicts the actual size of the file (80 bytes), while the size of **bar** is just the size of the name string **foo** (3 bytes). Such an alias is referred to as a *soft* link. It is possible to manipulate the contents of the file with equal privilege with either name. However, the difference arises in file deletion. Deletion of **foo** results in the removal of the file contents. The name **bar** still exists but its alias **foo** and the file contents do not. Trying to access the file named **bar** results in an access error.

One may wonder why the operating system may want to support two different aliasing mechanisms, namely, hard and soft links. The tradeoff is one of efficiency and usability. A file name that is a soft link immediately tells you the original file name, whereas a hard link obfuscates this important detail.

Therefore, soft links increase usability. On the other hand, every time the file system encounters a soft link it has to resolve the alias by traversing its internal data structures (namely, i-nodes in Unix). We will see shortly how this is done in the Unix file system. The hard link directly points to the internal representation of the original file name. Therefore, there is no time lost in name resolution and can lead to improved file system performance.

However, a hard link to a directory can lead to circular lists, which can make deletion operations of directories very difficult. For this reason, operating systems such as Unix disallow creating hard links to directories.

Writing to an existing file results in over-writing the contents in most operating systems (UNIX, Windows). However, such writes may create a new version of the file in a file system that supports versioning.

- **Access rights:** This attribute specifies *who* has access to a particular file and *what* kind of privilege each allowed user enjoys. The privileges generally provided on a file include: *read, write, execute, change ownership, change privileges*. Certain privileges exist at the individual user level (e.g., either the creator or the users of a file), while some other privileges exist only for the system administrator (**`root`** in UNIX and **`administrator`** in Windows). For example, in UNIX the owner of a file may execute the "change the permission mode of a file" command

```
chmod u+w foo    /* u stands for user;
                              w stands for write;
                              essentially this command
                              says add write access
                              to the user;

                         */
```

that gives write permission to the owner to a file named **`foo`**. On the other hand, only the system administrator may execute the "change ownership" command

```
chown rama foo
```

that makes **`rama`** the new owner of the file **`foo`**.

An interesting question that arises as a file system designer is deciding on the granularity of access rights. Ideally, we may want to give individual access rights to *each* user in the system to *each* file in the system. This choice results in an O (*n*) metadata space overhead per file, where *n* is the number of users in the system. Operating systems exercise different design choices to limit space overhead. For example, UNIX divides the world into three categories: *user, group, all. User* is an authorized user of the system; *group* is a set of authorized users; and *all* refers to all authorized users on the system. The system administrator maintains the membership of different group names. For example, students in a CS2200 class may all belong to a group name cs2200. UNIX supports the notion of individual ownership and group ownership to any file. The owner of a file may change the group ownership for a file using the command

```
chgrp cs2200 foo
```

that makes **`cs2200`** the group owner of the file **`foo`**.

With the world divided into three categories as above, UNIX provides *read, write, and execute* privilege for each category. Therefore, 3 bits encode the access rights for each category (1 each for read, write, execute). The execute privilege allows a file to be treated as an executable program. For example, the compiled and linked output of a compiler is a binary executable file. The following example shows all the visible metadata associated with a file in UNIX:

```
rwxrw-r--   1 rama     fac   2364 Apr 18 19:13 foo
```

The file `foo` is owned by rama, group-owned by fac. The first field gives the access rights for the three categories. The first three bits (rwx) give read, write, and execute privilege to the user (rama). The next three bits (rw-) give read and write privilege (no execute privilege) to the group (fac). The last three bits (r--) give read privilege (no write or execute privilege) to all users. The "1" after the access rights states the number of hard links that exists to this file. The file is of size 2364 bytes and the modification timestamp of the contents of the file is April 18 at 19:13 hours. Windows operating system and some flavors of UNIX operating systems allow a finer granularity of access rights by maintaining an *access control list (ACL)* for each file. This flexibility comes at the expense of increased metadata for each file.

Attribute	Meaning	Elaboration
Name	Name of the file	Attribute set at the time of creation or renaming
Alias	Other names that exist for the same physical file	Attribute gets set when an alias is created; system such as Unix provide explicit commands for creating aliases for a given file; Unix supports aliasing at two different levels (physical or hard, and symbolic or soft)
Owner	Usually the user who created the file	Attribute gets set at the time of creation of a file; systems such as Unix provide mechanism for the file's ownership to be changed by the superuser
Creation time	Time when the file was created first	Attribute gets set at the time a file is created or copied from some other place
Last write time	Time when the file was last written to	Attribute gets set at the time the file is written to or copied; in most file systems the creation time attribute is the same as the last write time attribute; Note that moving a file from one location to another preserves the creation time of the file
Privileges • **Read** • **Write** • **Execute**	The permissions or access rights to the file specifies who can do what to the file;	Attribute gets set to default values at the time of creation of the file; usually, file systems provide commands to modify the privileges by the owner of the file; modern file systems such NTFS provide an access control list (ACL) to give different levels of access to different users
Size	Total space occupied on the file system	Attribute gets set every time the size changes due to modification to the file

Table 11.1: File System Attributes

Unix command	Semantics	Elaboration
touch <name>	Create a file with the name <name>	Creates a zero byte file with the name <name> and a creation time equal to the current wall clock time
mkdir <sub-dir>	Create a sub-directory <sub-dir>	The user must have write privilege to the current working directory (if <sub-dir> is a relative name) to be able to successfully execute this command
rm <name>	Remove (or delete) the file named <name>	Only the owner of the file (and/or superuser) can delete a file
rmdir <sub-dir>	Remove (or delete) the sub-directory named <sub-dir>	Only the owner of the <sub-dir> (and/or the superuse) can remove the named sub-directory
ln –s <orig> <new>	Create a name <new> and make it symbolically equivalent to the file <orig>	This is name equivalence only; so if the file <orig> is deleted, the storage associated with <orig> is reclaimed, and hence <new> will be a dangling reference to a non-existent file
ln <orig> <new>	Create a name <new> and make it physically equivalent to the file <orig>	Even if the file <orig> is deleted, the physical file remains accessible via the name <new>
chmod <rights> <name>	Change the access rights for the file <name> as specified in the mask <rights>	Only the owner of the file (and/or the superuser) can change the access rights
chown <user> <name>	Change the owner of the file <name> to be <user>	Only superuser can change the ownership of a file
chgrp <group> <name>	Change the group associated with the file <name> to be <group>	Only the owner of the file (and/or the superuser) can change the group associated with a file
cp <orig> <new>	Create a new file <new> that is a copy of the file <orig>	The copy is created in the same directory if <new> is a file name; if <new> is a directory name, then a copy with the same name <orig> is created in the directory <new>
mv <orig> <new>	Renames the file <orig> with the name <new>	Renaming happens in the same directory if <new> is a file name; if <new> is a directory name, then the file <orig> is moved into the the directory <new> preserving its name <orig>
cat/more/less <name>	View the file contents	

Table 11.2: Summary of Common Unix file system commands

Table 11.1 summarizes common file system attributes and their meaning. Table 11.2 gives a summary of common commands available in most Unix file systems. All the commands are with respect to the current working directory. Of course, an exception to this rule is if the command specifies an absolute Unix path-name (e.g. /users/r/rama).

11.2 Design Choices in implementing a File System on a Disk Subsystem

We started discussion on file systems by presenting a file as a software abstraction for an input/output device. An equally important reason for file systems arises from the need to keep information around beyond the lifetime of a program execution. A file serves as a convenient abstraction for meeting this need. Permanent read/write storage is the right answer for keeping such persistent information around. File system is another important software subsystem of the operating system. Using disk as the permanent storage, we will discuss the design choices in implementing a file system.

As we have seen in Chapter 10, physically a disk consists of platters, tracks, and sectors. A given disk has specific fixed values for these hardware parameters. Logically, the corresponding tracks on the various platters form a cylinder. There are four components to the latency for doing input/output from/to a disk:

- Seek time to a specific cylinder
- Rotational latency to get the specific sector under the read/write head of the disk
- Transfer time from/to the disk controller buffer
- DMA transfer from/to the controller buffer to/from the system memory

We know that a file is of arbitrary size commensurate with the needs of the program. For example, files that contain simple ASCII text may be a few Kilobytes in size. On the other hand, a movie that you may download and store digitally on the computer occupies several hundreds of Megabytes. The file system has to bridge the mismatch between the user's view of a file as a storage abstraction, and physical details of the hard drive. Depending on its size, a file may potentially occupy several sectors, several tracks, or even several cylinders.

Therefore, one of the fundamental design issues in file system is the physical representation of a file on the disk. Choice of a design point should take into consideration both end users' needs and system performance. Let us understand these issues. From a user's perspective, there may be two requirements. One, the user may want to view the contents of a file sequentially (e.g. UNIX **`more`**, **`less`**, and **`cat commands`**). Two, the user may want to search for something in a file (e.g. UNIX **`tail`** command). The former implies that the physical representation should lend itself to efficient *sequential access* and the latter to *random access*. From the system performance point of view, the file system design should lend itself to easily *growing* a file when needed and for *efficient allocation* of space on the disk for new files or growth of existing files.

Therefore the *figures of merit*[1] for a file system design are:

- Fast sequential access
- Fast random access
- Ability to grow the file
- Easy allocation of storage
- Efficiency of space utilization on the disk

In the next few paragraphs, we will identify several file allocation schemes on a hard drive. For each scheme, we will discuss the data structures needed in the file system, and the impact of the scheme on the figures of merit. We will establish some common terminology for the rest of the discussion. An *address* on the disk is a triple {*cylinder#, surface#, sector#*}. The file system views the disk as consisting of *disk blocks*, a design parameter of the file system. Each disk block is a physically contiguous region of the disk (usually a set of sectors, tracks, or cylinders depending on the specific allocation scheme), and is the smallest granularity of disk space managed by the file system. For simplicity of the discussion we will use, *disk block address,* as a short hand for the disk address (the 4-tuple, {*cylinder#, surface#, sector#, size of disk block*}) corresponding to a particular disk block, and designate it by a unique integer.

11.2.1 Contiguous Allocation

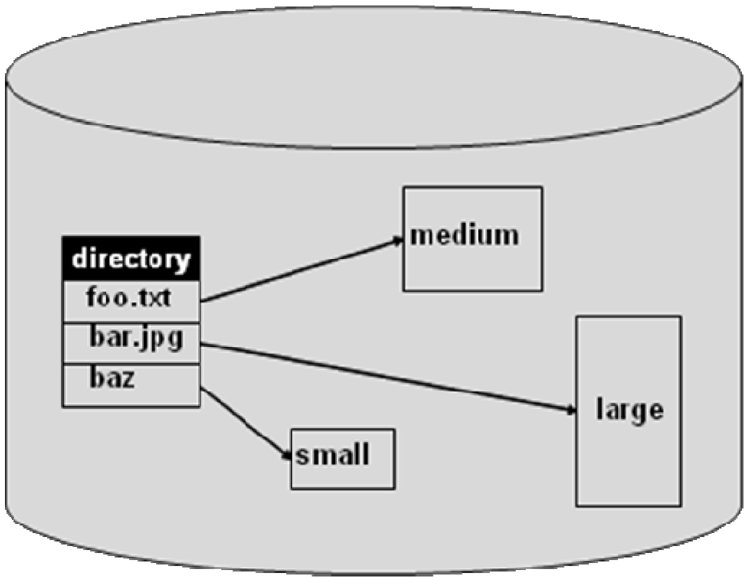


Figure 11.4 shows a disk with a directory that maps file name to a disk block address for contiguous allocation. The size of contiguous allocation matches the size of the file.

This disk allocation scheme has similarities to both the fixed and variable size partition based memory allocation schemes discussed in Chapter 8. At file creation time, the file system pre-allocates a fixed amount of space to the file. The amount of space allocated depends on the type of file (e.g., a text file versus a media file). Further, the amount of

[1] A figure of merit is a criterion used to judge the performance of the system.

space allocated is the maximum size that the file can grow to. Figure 11.4 shows the data structures needed for this scheme. Each entry in the *directory* data structure contains a mapping of the file name to the disk block address, and the number of blocks allocated to this file.

The file system maintains a *free list* of available disk blocks (Figure 11.5). Free list is a data structure that enables the file system to keep track of the currently unallocated disk blocks. In Chapter 8, we discussed how the memory manager uses the free list of physical frames to make memory allocation upon a page fault; similarly, the file system uses the free list of disk blocks to make disk block allocation for the creation of new files. As we will see, the details of this free list depend on the specific allocation strategy in use by the file system.

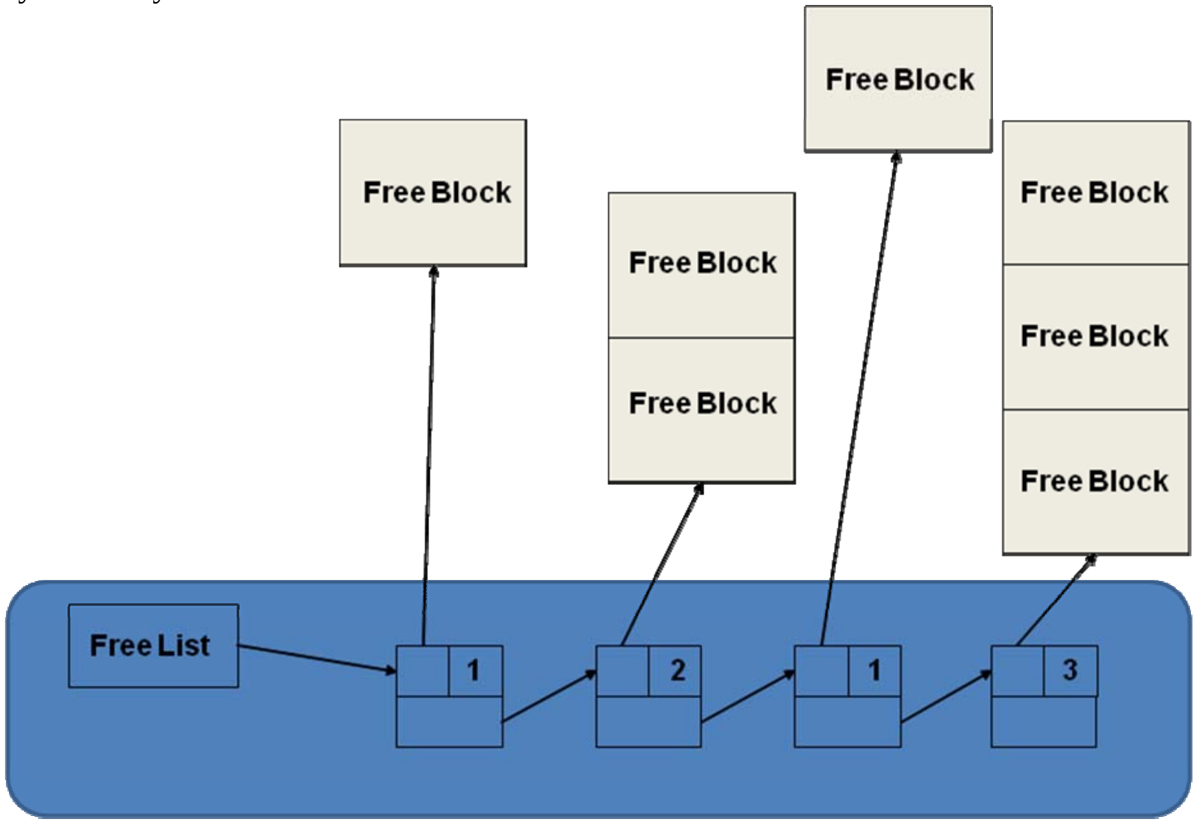


Figure 11.5 shows a free list, with each node containing {pointer to the starting disk block address, number of blocks} for contiguous allocation

For contiguous allocation, each node in this free list gives the starting disk block address and the number of available blocks. Allocation of disk blocks to a new file may follow one of *first fit* or *best fit* policy. Upon file deletion, the released disk blocks return to the free list. The file system combines adjacent nodes to form bigger contiguous disk block partitions. Of course, the file system does such *compaction* infrequently due to the overhead it represents. Alternatively, the file system may choose to do such compaction upon an explicit request from the user. Much of this description should sound familiar to the issues discussed in Chapter 8 for variable size memory partitions. Similar to that memory management scheme, this disk allocation strategy also suffers from *external fragmentation.* Further, since the file system commits a fixed size chunk of disk blocks (to allow for the maximum expected growth size of that file) at file creation time, this scheme suffers from *internal fragmentation* (similar to the fixed size partition memory allocation scheme discussed in Chapter 8).

The file system could keep these data structures in memory or on the disk. The data structures have to be in persistent store (i.e. some disk blocks are used to implement these data structures) since the files are persistent. Therefore, these data structures reside on the disk itself. However, the file system caches this data structure in memory to enable quick allocation decisions as well as to speed up file access.

Let us analyze this scheme qualitatively with respect to the figures of merit. Allocation can be expensive depending on the algorithm used (first fit or best fit). Since a file occupies a fixed partition (a physically contiguous region on the disk), both sequential access and random access to a file is efficient. Upon positioning the disk head at the starting disk block address for a particular file, the scheme incurs very little additional time for seeking different parts of the file due to the nature of the allocation. There are two downsides to the allocation scheme:

1. The file cannot grow in place beyond the size of the fixed partition allocated at the time of file creation. One possibility to get around this problem is to find a free list node with a bigger partition and copy the file over to the new partition. his is an expensive choice; besides, such a larger partition has to be available. The file system may resort to compaction to create such a larger partition.
2. As we said already, there is potential for significant wastage due to internal and external fragmentation.

Example 1:

Given the following:

Number of cylinders on the disk	=	10,000
Number of platters	=	10
Number of surfaces per platter	=	2
Number of sectors per track	=	128
Number of bytes per sector	=	256
Disk allocation policy	=	contiguous cylinders

(a) How many cylinders should be allocated to store a file of size 3 Mbyte?
(b) How much is the internal fragmentation caused by this allocation?

Answer:

(a)

Number of tracks in a cylinder = number of platters * number of surfaces per platter
= 10 * 2 = 20

Size of track = number of sectors in track * size of sector
= 128 * 256
= 2^{15} bytes

Capacity in 1 cylinder = number of tracks in cylinder * size of track
= $20 * 2^{15}$
= $10 * 2^{16}$ bytes

Number of cylinders to host a 3 MB file = CEIL $((3 * 2^{20}) / (10 * 2^{16}))$ **= 5**

(b)
Internal fragmentation = 5 cylinders – 3 MB
= 3276800-3145728 **= 131072 bytes**

11.2.2 Contiguous Allocation with Overflow Area

This strategy is the same as the previous one with the difference that the file system sets aside an *overflow* region that allows spillover of large files that do not fit within the fixed partition. The overflow region also consists of physically contiguous regions allocated to accommodate such spillover of large files. The file system needs an additional data structure to manage the overflow region. This scheme fares exactly similarly with respect to the figures of merit as the previous scheme. However, on the plus side this scheme allows file growth limited only by the maximum amount of space available in the overflow region without the need for any other expensive operation as described above. On the minus side, random access suffers slightly for large files due to the spill into the overflow region (additional seek time).

Despite some of the limitations, contiguous allocation has been used quite extensively in systems such IBM VM/CMS due to the significant performance advantage for file access times.

11.2.3 Linked Allocation

In this scheme, the file system deals with allocation at the level of individual disk blocks. The file system maintains a *free list* of all available disk blocks. A file occupies as many disk blocks as it takes to store it on the disk. The file system allocates the disk blocks from the free list as the file grows. The free list may actually be a linked list of the disk blocks with each block pointing to the next free block on the disk. The file system has the head of this list cached in memory so that it can quickly allocate a disk block to satisfy a new allocation request. Upon deletion of a file, the file system adds its disk blocks to the free list. In general having such a linked list implemented via disk blocks leads to expensive traversal times for free-list maintenance. An alternative is to implement the free list as a bit vector, one bit for each disk block. The block is free if the corresponding bit is 0 and busy if it is 1.

Note that the free list changes over time as applications produce and delete files, and as files grow and shrink. Thus, there is no guarantee that a file will occupy contiguous disk blocks. Therefore, a file is physically stored as a linked list of the disk blocks assigned to it as shown in Figure 11.6. As in the previous allocation schemes, some disk blocks hold the persistent data structures (free list and directory) of the file system.

On the plus side, the allocation is quick since it is one disk block at a time. Further, the scheme accommodates easy growth of files. There is no external fragmentation due to the on-demand allocation. Consequently, the scheme never requires disk compaction. On the minus side, since the disk blocks of a file may not be contiguous the scheme performs poorly for file access compared to contiguous allocation, especially for random access due to the need for retrieving next block pointers from the disk blocks. Even for sequential access, the scheme may be inefficient due to the variable seek time for the disk blocks that make up the list. The error-prone nature of this scheme is another downside: any bug in the list maintenance code leads to a complete breakdown of the file system.

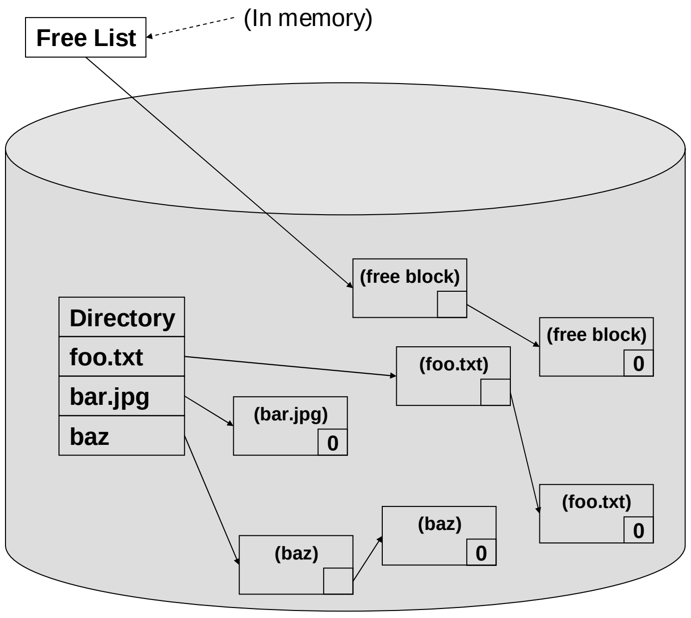


Figure 11.6: Linked Allocation

11.2.4 File Allocation Table (FAT)

This is a variation of the linked allocation. A table on the disk, *File Allocation Table (FAT)*, contains the linked list of the files currently populating the disk (see Figure 11.7). The scheme divides the disk logically into partitions. Each partition has a FAT, in which each entry corresponds to a particular disk block. The *free/busy* field indicates the availability of that disk block (0 for free; 1 for busy). The *next* field gives the next disk block of the linked list that represents a file. A distinguished value (-1) indicates this entry is the last disk block for that file. A single *directory* for the entire partition contains the *file name* to *FAT index* mapping as shown in Figure 11.7. Similar to the linked allocation, the file system allocates a disk block on demand to a file.

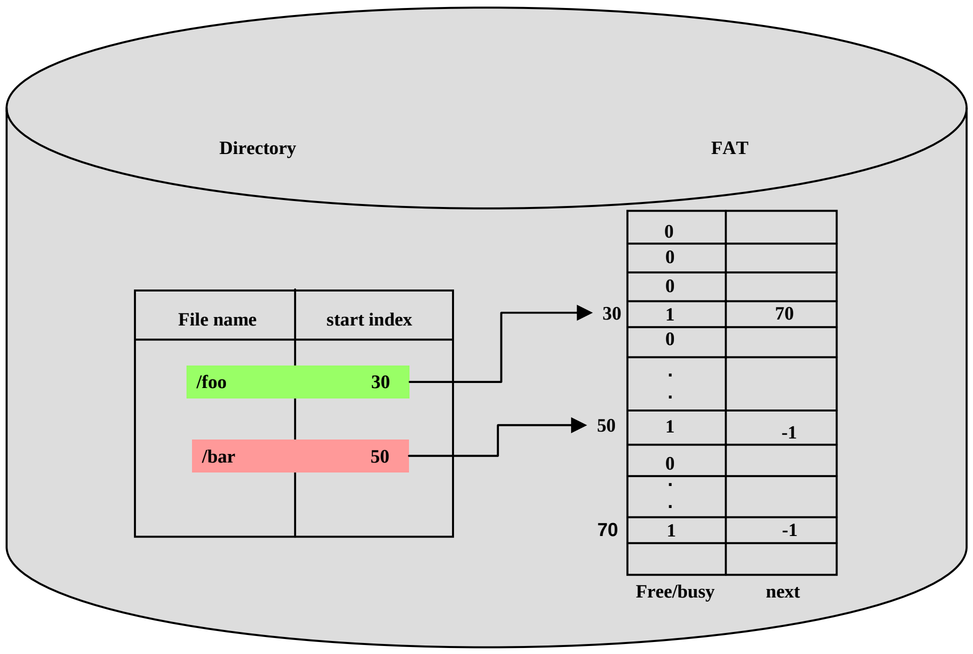


Figure 11.7: File Allocation Table (FAT)

For example, */foo* occupies two disk blocks: 30 and 70. The *next* field of entry 30 contains the value 70, the address of the next disk block. The next field of entry 70 contains -1 indicating this is the last block for */foo*. Similarly, */bar* occupies one disk block (50). If */foo* or */bar* were to grow, the scheme will allocate a free disk block and fix up the FAT accordingly.

Let us analyze the *pros* and *cons* of this scheme. Since FAT captures the linked list structure of the disk in a tabular data structure, there is less chance of errors compared to the linked allocation. By caching FAT in memory, the scheme leads to efficient allocation times compared to linked allocation. The scheme performs similar to linked allocation for sequential file access. It performs better than linked allocation for random access since the FAT contains the next block pointers for a given file.

One of the biggest downside to this scheme is the logical partitioning of a disk. This leads to a level of management of the space on the disk for the end user that is not pleasant. It creates artificial scarcity of space on the disk in a particular partition even when there is plenty of physical space on the disk. However, due to its simplicity (centralized data structures in the directory and FAT) this allocation scheme was popular in early personal computer operating systems such as MS-DOS and IBM OS/2.

Example 2:

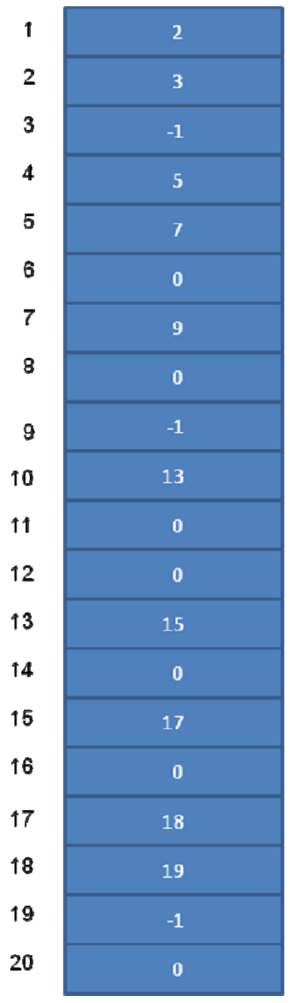
This question is with respect to the disk space allocation strategy referred to as **FAT.**
Assume there are 20 data blocks numbered 1 through 20.
There are three files currently on the disk:

foo occupies disk blocks 1, 2 and 3;
bar occupies disk blocks 10, 13, 15, 17, 18 and 19

gag occupies disk blocks 4, 5, 7 and 9

Show the contents of the **FAT** (show the free blocks and allocated blocks per convention used in this section).

Answer:



1	2
2	3
3	-1
4	5
5	7
6	0
7	9
8	0
9	-1
10	13
11	0
12	0
13	15
14	0
15	17
16	0
17	18
18	19
19	-1
20	0

11.2.5 Indexed Allocation

This scheme allocates an *index* disk block for each file. The index block for a file is a fixed-size data structure that contains addresses for data blocks that are part of that file. Essentially, this scheme aggregates data block pointers for a file scattered all over the FAT data structure (in the previous scheme) into one table per file as shown in Figure 11.8. This table called *index node* or *i-node*, occupies a disk block. The *directory* (also on the disk) contains the *file name* to *index node* mapping for each file. Similar to the linked allocation, this scheme maintains the *free list* as a bit vector of disk blocks (0 for free, 1 for busy).

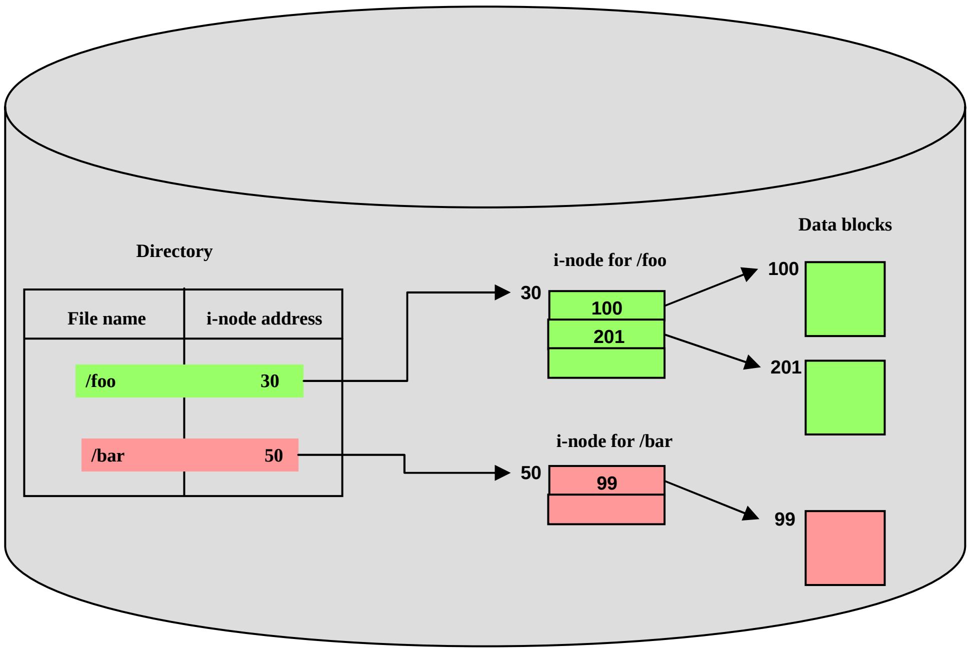


Figure 11.8: Indexed Allocation

Compared to FAT, this scheme performs better for random access since i-node aggregates all the disk block pointers into one concise data structure. The downside to this scheme is the limitation on the maximum size of a file, since i-node is a fixed size data structure per file with direct pointers to data blocks. The number of data block pointers in an i-node bounds the maximum size of the file.

We will explore other schemes that remove this restriction on maximum file size in the following subsections.

Example 3:

Consider an indexed allocation scheme on a disk

- The disk has 10 platters (2 surfaces per platter)
- There are 1000 tracks in each surface
- Each track has 400 sectors
- There are 512 bytes per sector
- Each i-node is a fixed size data structure occupying one sector.
- A data block (unit of allocation) is a contiguous set of 2 cylinders
- A pointer to a disk data block is represented using an 8-byte data structure.

a) What is the minimum amount of space used for a file on this system?

b) What is the maximum file size with this allocation scheme?

Answer:

Size of a track = number of sectors per track * size of sector

= 400 * 512 bytes = 200 Kbytes (K = 1024)

Number of tracks in a cylinder = number of platters * number of surfaces per platter

= 10 * 2 = 20

Size of a cylinder = number of tracks in a cylinder * size of track
= 20 * 200 Kbytes
= 4000 Kbytes

Unit of allocation (data block) = 2 cylinders
= 2 * 4000 Kbytes
= 8000 Kbytes

Size of i-node = size of sector = 512 bytes

a)

Minimum amount of space for a file = size of i-node + size of data block
= 512 + (8000 * 1024) bytes
= **8,192,512 bytes**

Number of data block pointers in an i-node = size of i-node / size of data block pointer
= 512/8 = 64

b)

Maximum size of file = Number of data block pointers in i-node * size of data block
= 64 * 8000 K bytes (K = 1024)
= **5,24,288,000 bytes**

11.2.6 Multilevel Indexed Allocation

This scheme fixes the limitation in the indexed allocation by making the i-node for a file an indirection table. For example, with one level indirection, each i-node entry points to a first level table, which in turn points to the data blocks as shown in Figure 11.9. In this figure, the i-node for foo contains pointers to a first level indirection index blocks. The number of first level indirection tables equals the number of pointers that an i-node can hold. These first level indirection tables hold pointers to the data blocks of the file.

The scheme can be extended to make an i-node a two (or even higher) level indirection table depending on the size of the files that need to be supported by the file system. The downside to this scheme is that even small files that may fit in a few data blocks have to go through extra levels of indirection.

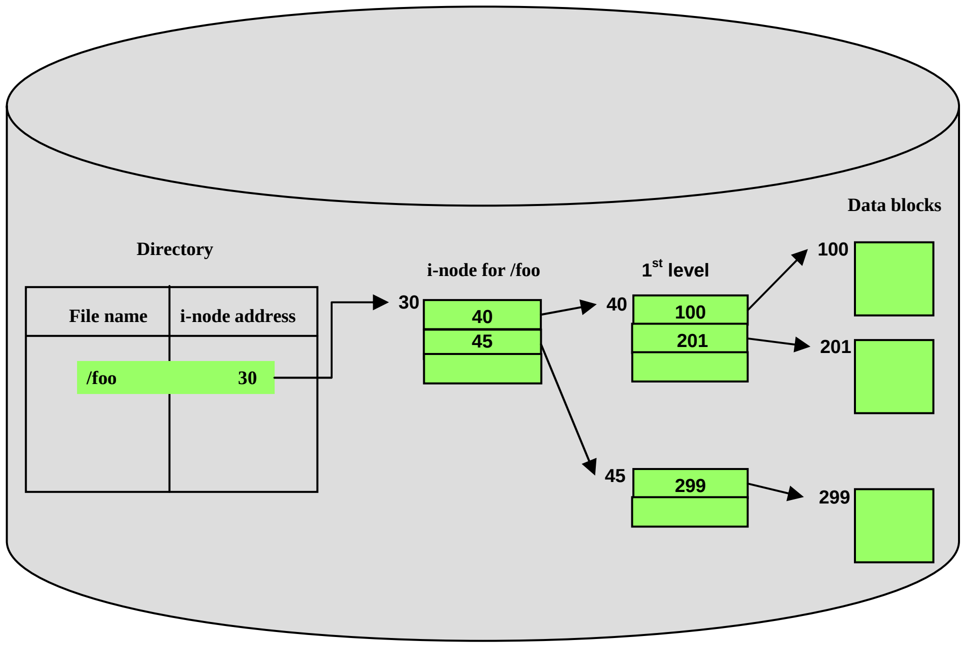


Figure 11.9: Multilevel Indexed Allocation (with one level of indirection)

11.2.7 Hybrid Indexed Allocation

This scheme combines the ideas in the previous two schemes to form a hybrid scheme. Every file has an i-node as shown in Figure 11.10. The scheme accommodates all the data blocks for small file with direct pointers. If the file size exceeds the capacity of direct blocks, then the scheme uses single or more levels of indirection for the additional data blocks. Figure 11.10 shows */foo* using direct, single indirect, and double indirect pointers. The i-node for */foo* is a complex data structure as shown in the figure. It has pointers to two direct data blocks (100 and 201); one pointer to an indirect index block (40); one pointer to a double indirect index block (45) which in turn has pointers to single indirect index blocks (60 and 70); and one pointer to a triple indirect index block (currently unassigned). The scheme overcomes the disadvantages of the previous two schemes while maintaining their good points. When a file is created with this scheme only the i-node is allocated for it. That is, the single, double, and triple indirect pointers are initially null. If the file size does not exceed the amount of space available in the direct blocks then there is no need to create additional index blocks. However, as the file grows and exceeds the capacity of the direct data blocks, space is allocated on a need basis for the first level index block, second level index block, and third level index block.

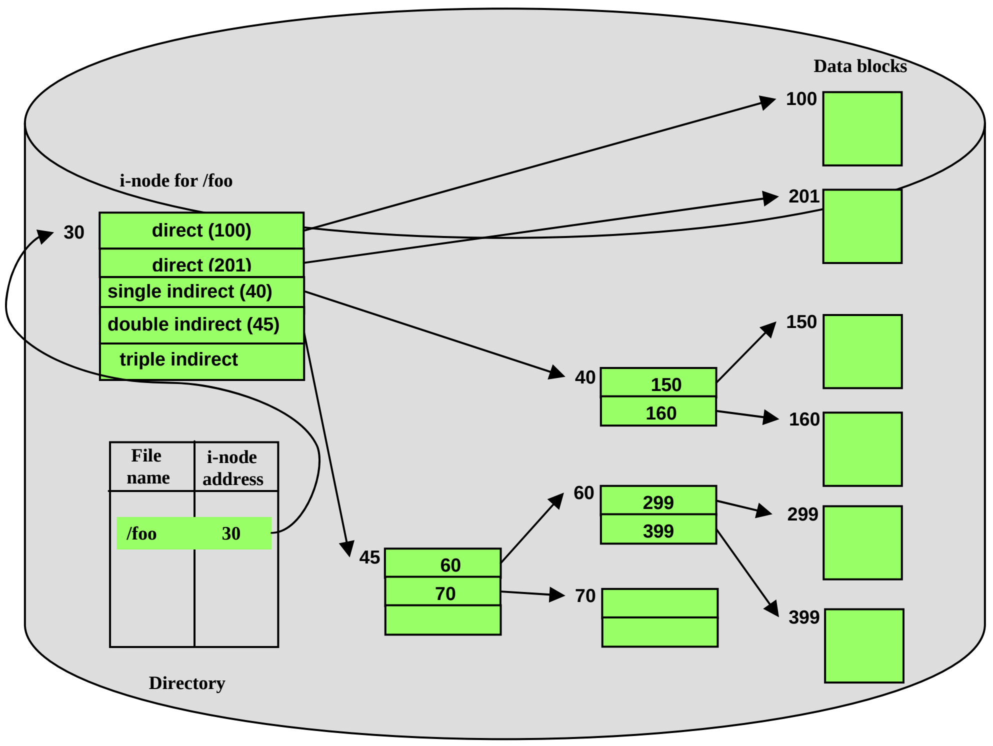


Figure 11.10: Hybrid Indexed Allocation

Example 4:

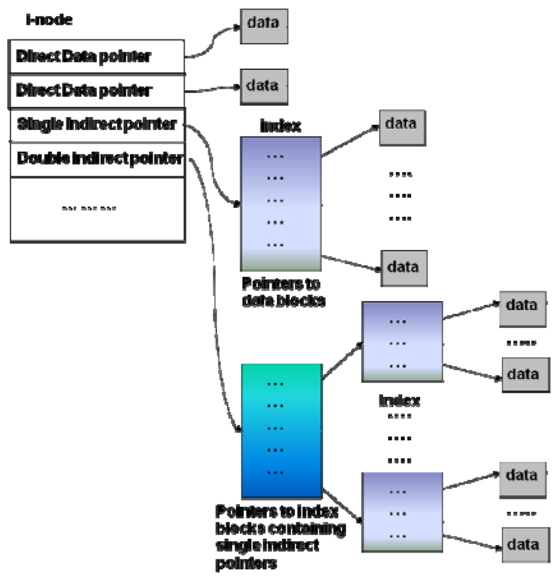
Given the following:

- Size of index block = 512 bytes
- Size of Data block = 2048 bytes
- Size of pointer = 8 bytes (to index or data blocks)

The i-node consists of

- 2 direct data block pointers,
- 1 single indirect pointer, and
- 1 double indirect pointer.

An index block is used for the i-node as well as for the index blocks that store pointers to other index blocks and data blocks. Pictorially the organization is as shown in the figure on the right. Note that the index blocks and data blocks are allocated on a need basis.



(a) What is the maximum size (in bytes) of a file that can be stored in this file system?
(b) How many data blocks are needed for storing the same data file of 266 KB?
(c) How many index blocks are needed for storing a data file of size 266 KB?

Answer:

(a)

Number of single indirect or double indirect pointers in an index block
= 512/8
= 64

Number of direct data blocks = 2

An i-node contains 1single indirect pointer that points to an index block (we call this *single indirect index block*) containing pointers to data blocks.
Number of data blocks with one level of indirection
= number of data block pointers in an index block
= 64

An i-node contains 1 double indirect pointer that points to an index block (we call this *double indirect index block*) containing 64 single indirect pointers. Each such pointer point to a single indirect index block that in turn contains 64 pointers to data blocks. This is shown in the figure.

Number of data block with two levels of indirection
= number of single indirect pointers in an index block * number of data block pointers index node
= 64*64

Max file size in blocks
= number of direct data blocks +
number of data blocks with one level of indirection +
number of data blocks with two levels of indirection
= 2 + 64 + 64*64 = 4162 data blocks

Max file size = Max file size in blocks * size of data block
= 4162 * 2048 bytes
= 8,523,776 bytes

(b)

Number of data blocks needed
= size of file / size of data block
= $266 * 2^{10} / 2048$
= **133**

(c)

To get 133 data blocks we need:

1 i-node	(gets us 2 direct data blocks)
1 single indirect index block	(gets us 64 data blocks)
1 double indirect index block	
2 single indirect index blocks off the double indirect index block	(gets us the remaining 64 + 3 data blocks)

Therefore, totally we need **5 index blocks**

11.2.8 Comparison of the allocation strategies

Table 11.3 summarizes the relative advantages and disadvantages of the various allocation strategies.

Allocation Strategy	File representation	Free list maintenance	Sequential Access	Random Access	File growth	Allocation Overhead	Space Efficiency
Contiguous	Contiguous blocks	complex	Very good	Very good	messy	Medium to high	Internal and external fragmentation
Contiguous With Overflow	Contiguous blocks for small files	complex	Very good for small files	Very good for small files	OK	Medium to high	Internal and external fragmentation
Linked List	Non-contiguous blocks	Bit vector	Good but dependent on seek time	Not good	Very good	Small to medium	Excellent
FAT	Non-contiguous blocks	FAT	Good but dependent on seek time	Good but dependent on seek time	Very good	Small	Excellent
Indexed	Non-contiguous blocks	Bit vector	Good but dependent on seek time	Good but dependent on seek time	limited	Small	Excellent
Multilevel Indexed	Non-contiguous blocks	Bit vector	Good but dependent on seek time	Good but dependent on seek time	Good	Small	Excellent
Hybrid	Non-	Bit vector	Good but	Good	Good	Small	Excellent

	contiguous blocks		dependent on seek time	but dependent on seek time			

Table 11.3: Comparison of Allocation Strategies

For the integrity of the file system across power failures, the persistent data structures of the file system (e.g., i-nodes, free list, directories, FAT, etc.) have to reside on the disk itself. The number of disk blocks devoted to hold these data structures represents space overhead of the particular allocation strategy. There is a time penalty as well for accessing these data structures. File systems routinely cache these critical data structures in main memory to avoid the time penalty.

11.3 Putting it all together

As a concrete example, UNIX operating system uses a hybrid allocation approach with hierarchical naming. With hierarchical naming, the directory structure is no longer centralized. Each i-node represents a part of the multipart hierarchical name. Except for the leaf nodes, which are data files, all the intermediate nodes are directory nodes. The type field in the i-node identifies whether this node is a directory or a data file. The i-node data structure includes fields for the other file attributes we discussed earlier such as access rights, timestamp, size, and ownership. In addition, to allow for aliasing, the i-node also includes a *reference count* field.

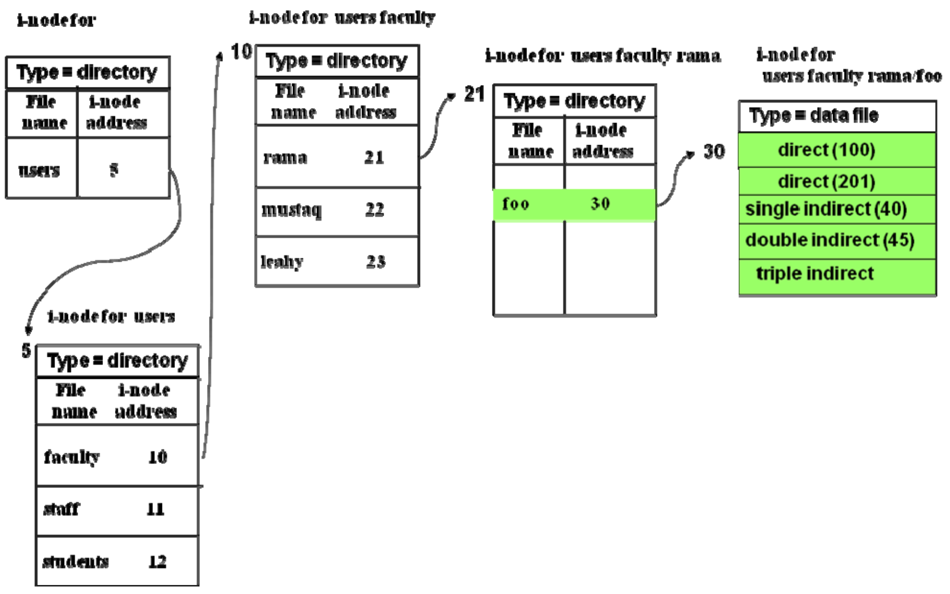


Figure 11.11: A simplified i-node structure for a hierarchical name in UNIX (a sequence of commands have resulted in creating the file /users/faculty/rama/foo)

Figure 11.11 shows the complete i-node structure for a file */users/faculty/rama/foo*. For the sake of simplicity, we do not show the data blocks.

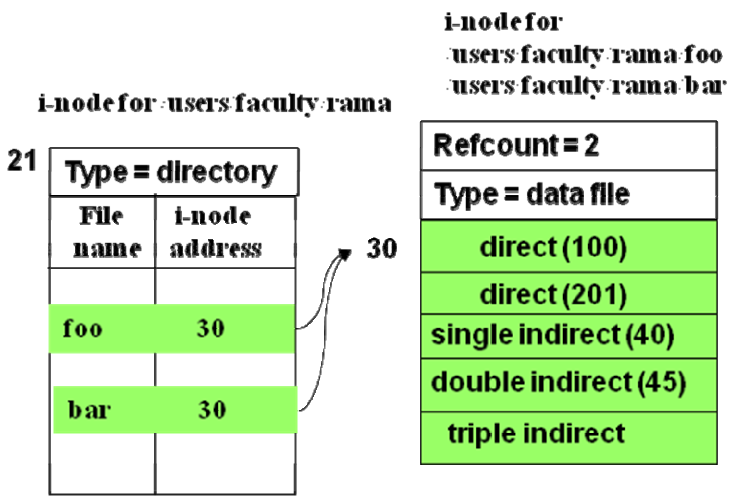


Figure 11.12: Two files foo and bar share an i-node since they are hard links to each other (the command "ln foo bar" results in this structure)

Figure 11.12 shows the i-node structure with *bar* as a hard link to *foo*. Both the files share the same i-node. Hence the *reference count* field in the i-node is 2. Figure 11.13 shows the i-node structure when a third file *baz* is symbolically linked to the name */users/faculty/rama/foo*. The i-node for *baz* denotes it as a symbolic link and contains the name */users/faculty/rama/foo*. The file system starts traversing the i-node structure given by the symbolic name (i.e. in this case, upon accessing *baz,* the traversal will start from the i-node for */*).

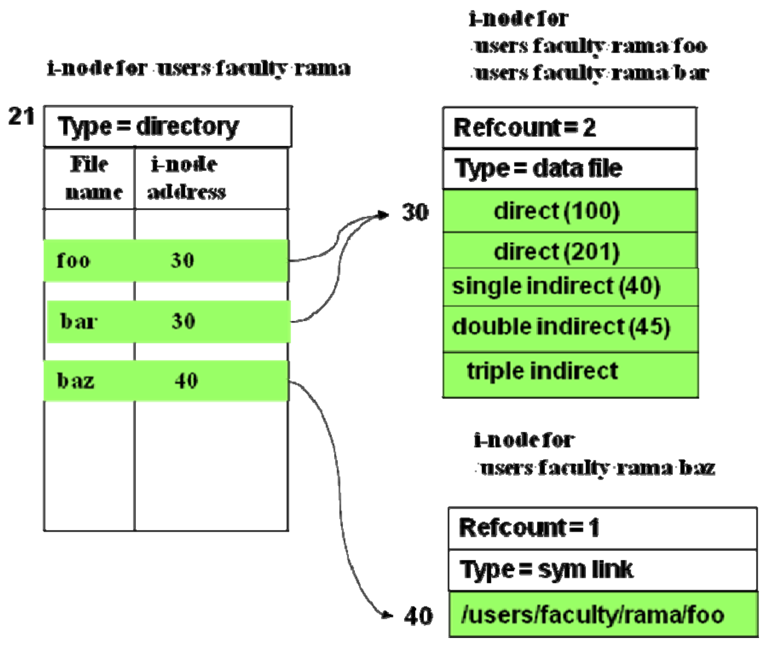


Figure 11.13: File baz is a symbolic link to /users/faculty/rama/foo

(the command “ln –s /users/faculty/rama/foo baz” results in this structure)

Example 5:

Current directory	/tmp
I-node for /tmp	20

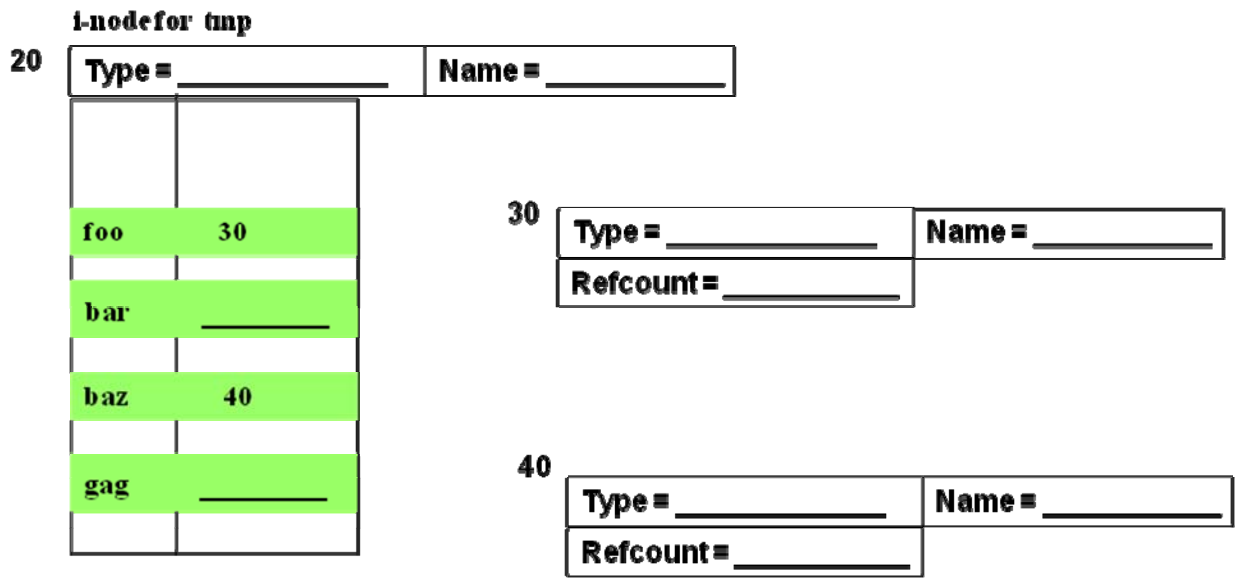
The following Unix commands are executed in the current directory:

```
     touch foo                    /*   creates a zero byte
file
                                            in the current
directory                                        */
     ln foo bar                   /*   create a hard link
               */
     ln –s /tmp/foo baz  /*   create a soft link
     */
     ln baz gag                   /*   create a hard link
               */
```

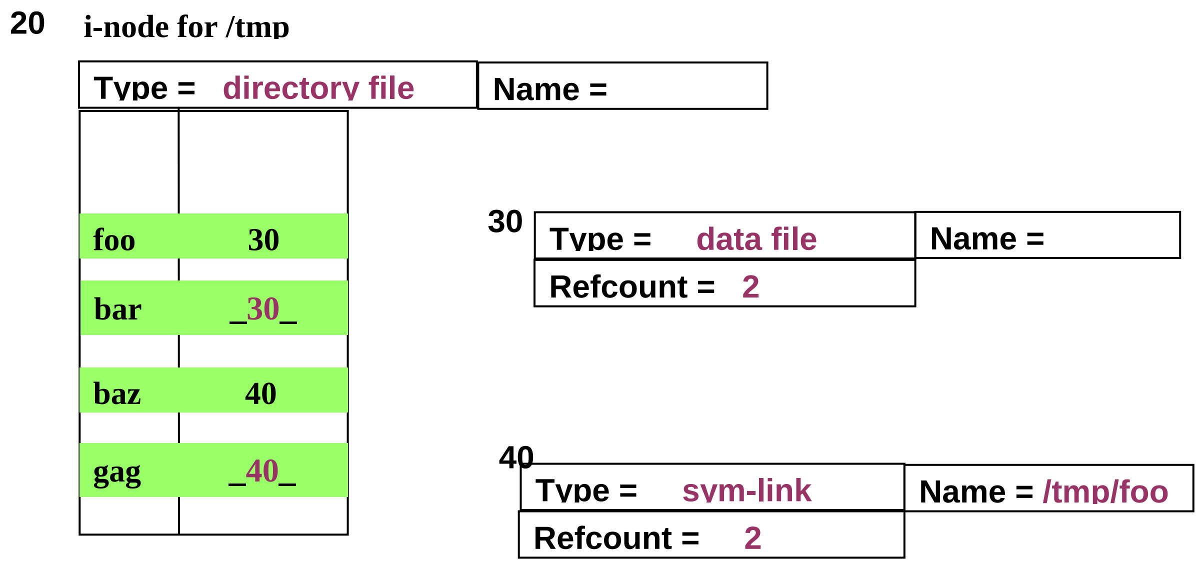
Note:

- Type of i-node can be one of directory-file, data-file, sym-link
  - ➢ if the type is sym-link then you have to give the name associated with that sym-link; otherwise the name field in the i-node is blank
- reference count is a non-zero positive integer

In the picture below fill in all the blanks to complete the contents of the various i-nodes.



Answer:



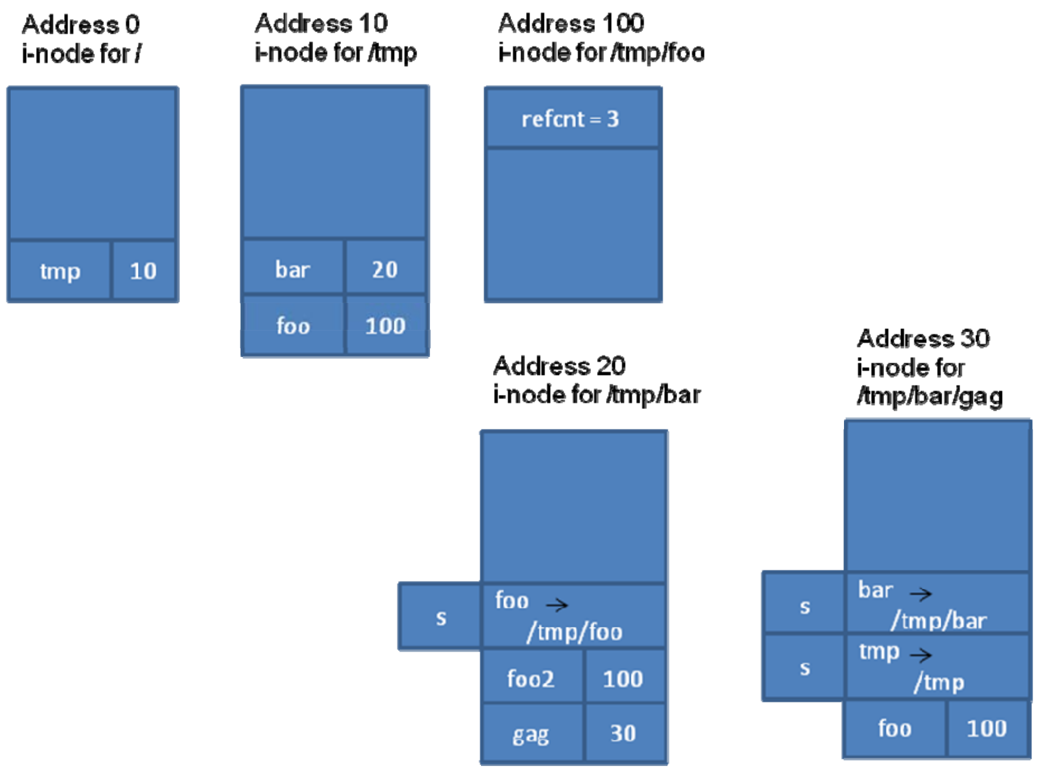
Example 6:

Given the following commands pictorially show the i-nodes and their contents. You can fabricate disk block addresses for the i-nodes to make the pictures simpler. Where relevant show the reference count for the i-nodes.

```
touch /tmp/foo
mkdir /tmp/bar
mkdir /tmp/bar/gag
ln /tmp/foo /tmp/bar/foo2
ln -s /tmp/foo /tmp/bar/foo
ln /tmp/foo /tmp/bar/gag/foo
ln -s /tmp/bar /tmp/bar/gag/bar
ln -s /tmp /tmp/bar/gag/tmp
```

Note:
mkdir creates a directory; touch creates a zero
byte file; ln is link command (-s denotes symbolic link).
Assume that the above files and directories are the only ones in the file system

Answer



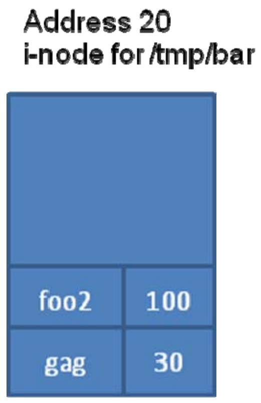
If we now execute the command:

```
rm /tmp/bar/foo
```

Show the new contents of the i-nodes. (SHOW ONLY THE AFFECTED I-NODES).

Answer:

`Only i-node for /tmp/bar changes as shown below.  The symbolic link entry is removed from the i-node.`



With reference to Figure 11.13, first let us understand what would happen to the i-node structure if the file *bar* was deleted. The entry bar will be removed from the i-node for */user/faculty/rama* and the *reference count* field in the i-node for */users/faculty/rama/foo* (block 30) will be decremented by 1 (i.e. the new *refcount* will be 1).

Next, let us see what would happen to the i-node structure once both *bar* and *foo* are deleted. Figure 11.14 depicts this situation. Since the *refcount* for i-node at block 30 goes to 0 with the deletion of both *foo* and *bar*, the file system returns the disk block 30 to the free list. Notice that *baz* still exists as a file in */users/faculty/rama*. This is because the file system only checks if the name being aliased at the time of creation of the symbolic link is valid. However, notice that no information about the symbolic link is kept in the i-node of the name ("foo" in this case). Therefore, there is no way for the file system to check for existence of symbolic links to a name at the time of deleting a name ("foo" in this case). However, any attempt to examine the contents of *baz* results in an error since the name */users/faculty/rama/foo* is non-existent so far as the file system is concerned. This shows illustrates the need for exercising care while using symbolic links.

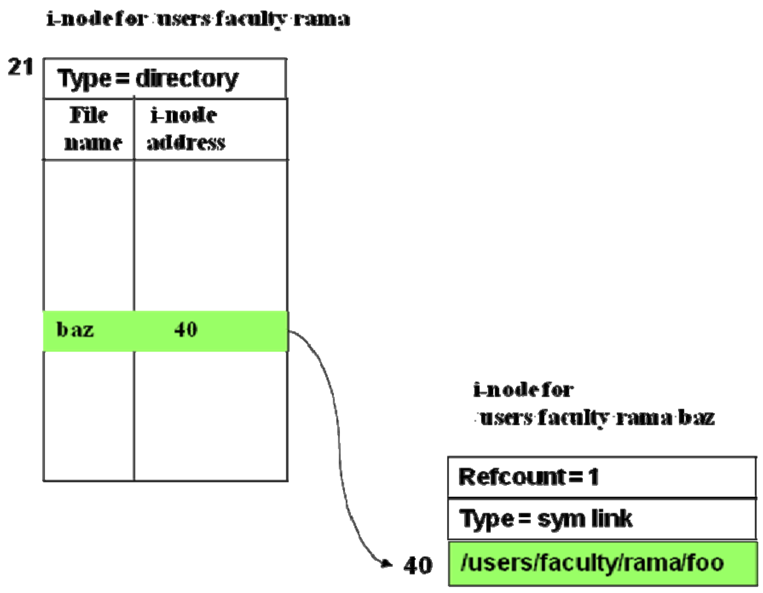


Figure 11.14: State of i-node structure after both foo and bar are deleted. (the commands "rm foo bar" results in this structure)

Example 7:

Consider the following:

```
  touch foo;        /* creates a zero byte file */
  ln foo bar;       /* creates a hard link called bar
                         to foo */
  ln -s foo baz;    /* creates a soft link called baz
                         to foo   */
  rm foo;           /* removes the file foo */
```

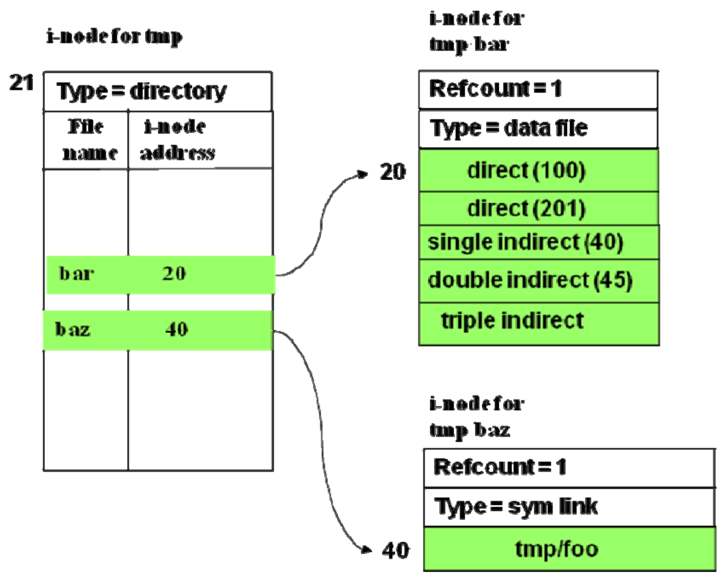
What would happen if we now execute the command

```
  cat baz;          /* attempt to view the content of baz
                     */
```

Answer:
Let the current directory be tmp where all the action shown above takes place. In this case, foo, bar, and baz are names created in the i-node for tmp.

Let the i-node for foo be 20. When we create the hard link bar, the refcount for i-node 20 goes up to 2. When foo is removed, the refcount for i-node drops to 1 but the i-node is not removed since bar has a hard link to it; however, the name foo is removed from the i-node of the current directory.

Therefore, when we attempt to view the contents of baz using the cat command, the file system flags an error since the name foo does not exist in the current directory any more. This is shown in the following figure:



11.3.1 i-node

In Unix, every file has a *unique* number associated with it, called the *i-node* number. You can think of this number as an *index* into a table that has all the information associated with the file (owner, size, name, etc.). As Unix systems evolved the complexity of the information associated with each file increased as well (the name itself can be arbitrarily long, etc.). Thus in modern Unix systems, each file is represented by a unique i-node data structure that occupies an entire disk block. The collection of i-nodes thus forms a logical table and the i-node number is simply the unique disk block address that indexes into this logical table and contains the information about that particular file. For the sake of convenience in implementing the file system, all the i-nodes occupy spatially adjacent disk blocks in the layout of the file system on the physical media. Typically, the file system reserves a sufficiently large number of disk blocks as i-nodes.

This implicitly becomes the limit of the maximum number of files in a given file system. It is customary for the storage allocation algorithm to maintain a bit vector (one bit for each i-node) that indicates whether a particular i-node is currently in use or not. This allows for efficient allocation of i-nodes upon file creation requests. Similarly, the storage manager may implement the free list of data blocks on the disk as a bit vector (one bit to signify the use or otherwise of a data block on the media) for efficient storage allocation.

11.4 Components of the File System

While it is possible to implement file systems at the user level, typically it is part of the operating system. Figure 11.15 shows the layers of the file system for the disk. We refer to the part of the operating system that manages the file system as the *File System (FS) Manager*. We break down the FS manager into the following layers for the sake of exposition.

- **Media independent layer**: This layer consists of the user interface, i.e. the *Application Program Interface (API)* provided to the users. The API module gives the file system commands for the user program to open and close files, read and write files, etc. This layer also consists of the *Name Resolver* module that translates the user-supplied name to an internal representation that is meaningful to the file system. For instance, the name resolver will be able to map the user specific name (e.g., E:\myphotos) to the specific device on which the file exists (such as the disk, CD, and Flash drive).
- **Media specific storage space allocation layer**: This layer embodies the space allocation (on file creation), space reclamation (on file deletion), free-list maintenance, and other functions associated with managing the space on the physical device. For instance, if the file system exists on disk, then the data structures and algorithms will reflect the discussions in Section 11.2.
- **Device driver**: This is the part that deals with communicating the command to the device and effecting the data transfer between the device and the operating system buffers. The device driver details (see Chapter 10 for a general description of device drivers) depend on the specifics of the mass storage that hosts the file system.
- **Media specific requests scheduling layer**: This layer is responsible for scheduling the requests from the OS commensurate with the physical properties of the device. For example, in the case of a disk, this layer may embody the disk scheduling algorithms that we learned in Chapter 10. As we observed in Chapter 10, even in the case of a disk, the scheduling algorithms may in fact be part of the device controller that is part of the drive itself. The scheduling algorithm may be quite different depending on the nature of the mass storage device.

There will be distinct instance of the bottom three layers of the software stack shown in Figure 11.15 for each mass storage device that provides a file system interface to the user. Thus, file is a powerful abstraction for hiding the details of the actual physical storage on which the data pertaining to the file is kept.

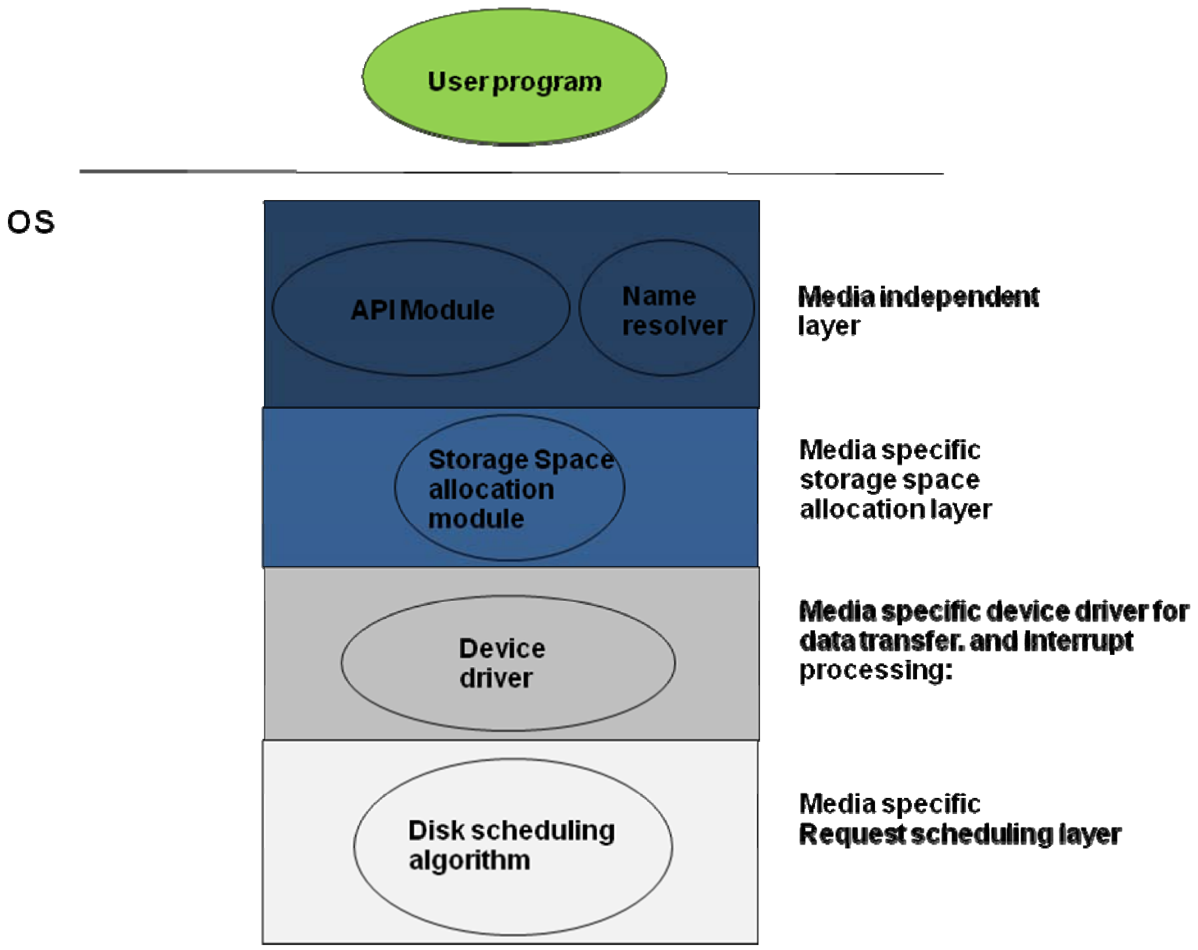

User program
OS
API Module
Name resolver
Media independent layer
Storage Space allocation module
Media specific storage space allocation layer
Device driver
Media specific device driver for data transfer. and Interrupt processing:
Disk scheduling algorithm
Media specific Request scheduling layer

Figure 11.15: Layers of a disk-specific file system manager

11.4.1 Anatomy of creating and writing files

As a concrete example, let us say your program makes an I/O call to create a file on the hard disk. The following steps trace the path of such an I/O call through the software layers shown in Figure 11.15.

1. The API routine for creating a file calls validates the call by checking the permissions, access rights, and other related information for the call. After such validation, it calls the name resolver.
2. The name resolver contacts the storage allocation module to allocate an i-node for the new file.
3. The storage allocation module gets a disk block from the free list and returns it to the name resolver. The storage allocation module will fill in the i-node commensurate with the allocation scheme (see Section 11.2). Let us assume the allocation in effect is the hybrid scheme (Section 11.2.7). Since the file has been created without any data as yet, no data blocks would have been allocated to the file.
4. The name resolver creates a directory entry and records the name to i-node mapping information for the new file in the directory.

Notice that these steps do not involve actually making a trip to the device since the data structures accessed by the file system (directory and free list) are all in memory.

Now let us say, your program writes to the file just created. Let us trace the steps through the software layers for this operation.

1. As before the API routine for file write will do its part in terms of validating the request.
2. The name resolver passes the memory buffer to the storage allocation module along with the i-node information for the file.
3. The storage allocation module allocates data blocks from the free list commensurate with the size of the file write. It then creates a request for disk write and hands the request to the device driver.
4. The device driver adds the request to its request queue. In concert with the disk-scheduling algorithm, the device driver completes the write of the file to the disk.
5. Upon completion of the file write, the device driver gets an interrupt from the disk controller that is passed back up to the file system, which in turn communicates with the CPU scheduler to continue the execution of your program from the point of file write.

It should be noted that as far as the OS is concerned, the file write call is complete as soon as the request is handed to the device driver. The success or failure of the actual call will be known later when the controller interrupts. The file system interacts with the CPU scheduler in exactly the same manner, as did the memory manager. The memory manager, in order to handle a page fault, makes an I/O request on behalf of the faulting process. The memory manager gets notified when the I/O completes so that it can tell the CPU scheduler to resume execution of the faulting process (see Chapter 8, Section 8.2). This is exactly what happens with the file system as well.

11.5 Interaction among the various subsystems

It is interesting to review the various software subsystems we have come across within the operating system so far: CPU scheduler, VM manager, File System (FS) Manager, and device drivers for the various I/O devices.

All of these subsystems are working for the user programs of course. For example, the VM manager is servicing page faults that a user program incurs implicitly performing I/O on behalf of the faulting process. The FS manager explicitly performs I/O catering to the requests of a user program to read and write files. In both cases, some mass storage device (hard disk, Flash memory, CD, DVD, etc.) is the target of the I/O operation. The device driver pertinent to the specific I/O request performs the I/O request and reports the result of the I/O operation back to the requestor.

The device driver has to figure out somehow whose I/O request was completed upon an I/O completion interrupt from the device controller. One simple and unifying method is to use the PCB data structure that we introduced in Chapter 6 (Section 6.4) as a communication vehicle among the subsystems.

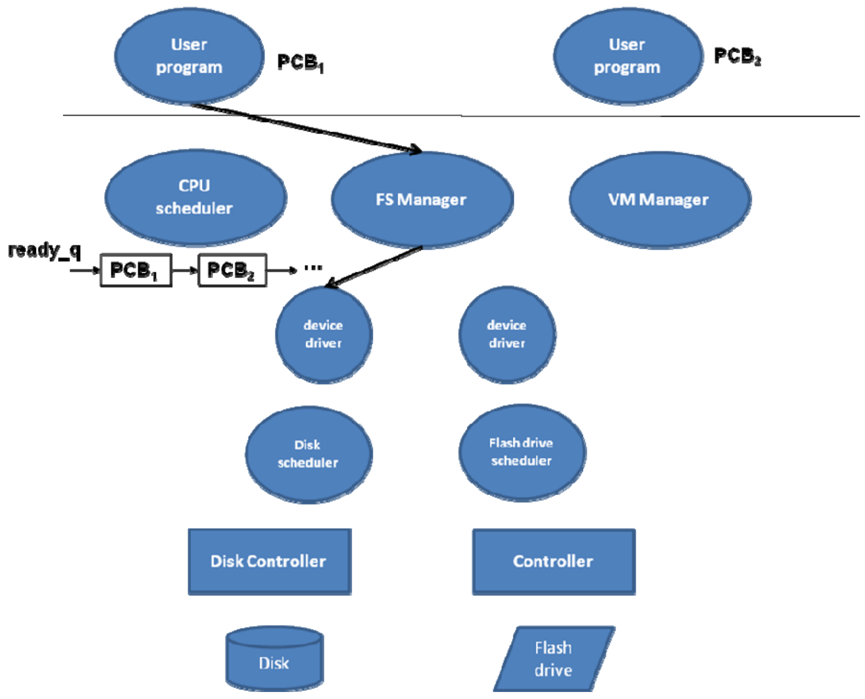


Figure 11.16-(a): Information flow on a file system call

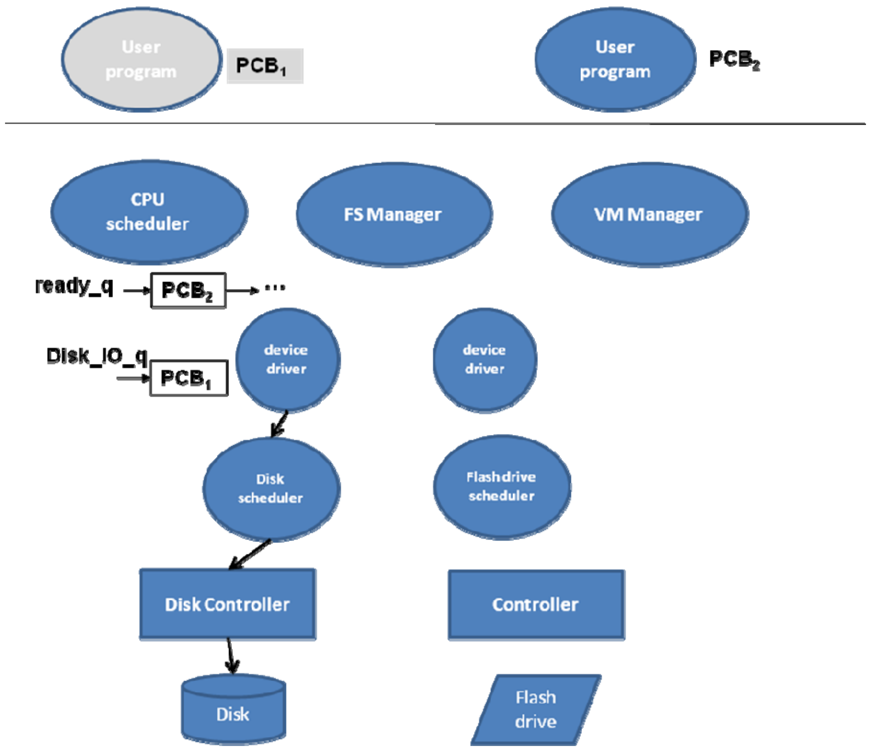


Figure 11.16-(b): Disk Driver handling the file system call from PCB$_1$

For example, the top layer of the FS manager passes the file I/O request to the device driver via the PCB of the process. This is straightforward and similar to a procedure call, except that the procedure call is across layers of the software stack shown in Figure 11.15. This information flow is shown in Figure 11.16-(a). Let PCB_1 represent the PCB of the process that makes the file system call. Once the call has been handed over to the device driver, the process is no longer runnable and this is depicted in Figure 11.16-(b). PCB_1 is now in the Disk_IO_q while being serviced by the device driver.

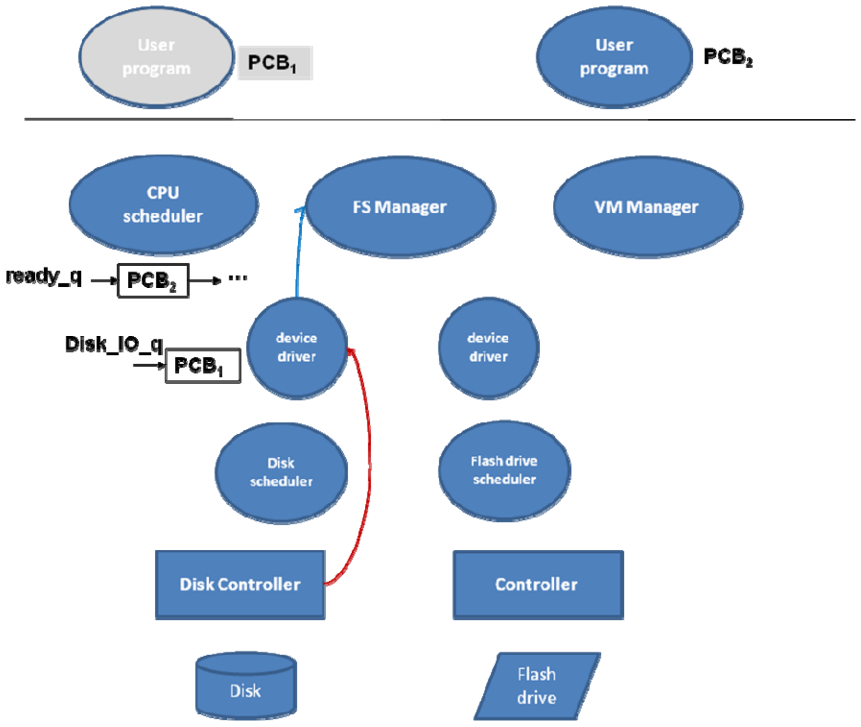


Figure 11.16-(c): A series of upcalls upon completion of the file I/O request (the red arrow is in hardware; the blue arrow is in software)

The device driver has to notify the upper layers of the system software stack upon completion of the I/O. In Chapter 6 (see Section 6.7.1.1), we introduced the concept of an *upcall*. This is essentially a mechanism for a lower layer in the system software stack to call a function in the upper layers. In fact, there is a continuum of upcalls as can be seen in Figure 11.16-(c). First, the disk controller makes an upcall in the form of an interrupt to the interrupt handler in the device controller. The interrupt handler knows exactly the process on whose behalf this I/O was initiated (from the Disk_IO_q). Of course, the interrupt handler in the device driver needs to know exactly who to call. For this reason, every upper layer in the system stack registers a handler with the lower layers for enabling such upcalls. Using this handler, the device driver makes an upcall to the FS manager to indicate completion of the file I/O request. One can see the similarity to this upcall mechanism and the way hardware interrupts are handled by the CPU (see Chapter 4). Due to the similarity both in terms of the function (asynchronously communicating events to the system) as well as the mechanism used to convey such events, upcalls are often referred to as *software interrupts*.

Upon receiving this upcall, the FS manager restores the PCB of the requesting process (PCB_1) back into the ready_q of the CPU scheduler as shown in Figure 11.16-(d). As can be seen, the interactions among the components of the operating system are enabled smoothly through the abstraction of the executing program in the form of a PCB. This is the power of the PCB abstraction.

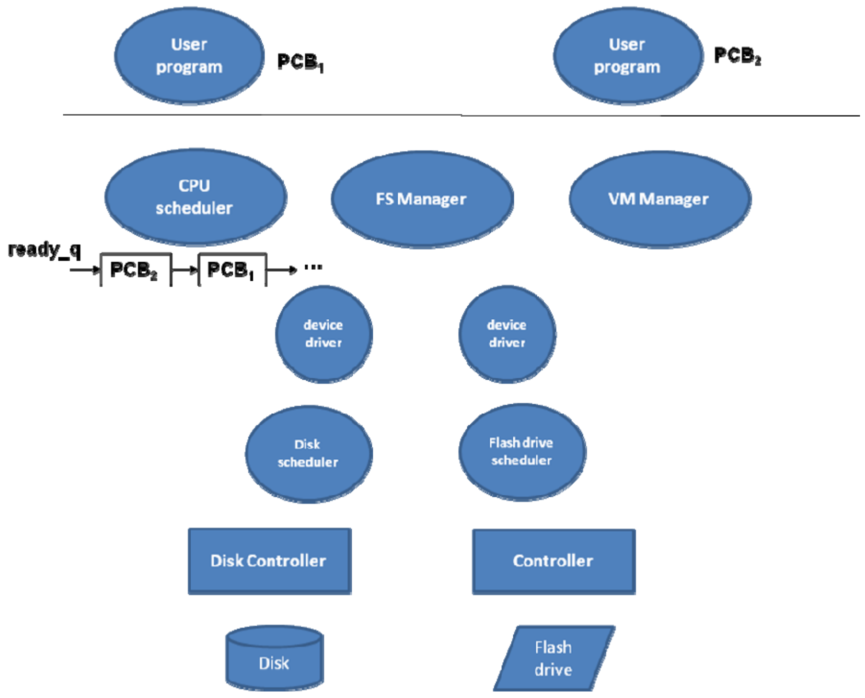


Figure 11.16-(d): FS Manager puts the process (PCB_1) that made the I/O call back in the CPU ready_q

A similar sequence of events would take place upon a page fault by a process; the only difference being that the VM Manager will be the initiator of the actions up and down the software stack shown in Figures 11.16-(a-d).

11.6 Layout of the file system on the physical media

Let us now understand how the operating system takes control of the resources in the system on power up.

In Chapter 10, we mentioned the basic idea behind booting an operating system. We mentioned how the BIOS performs the basic initialization of the devices and hands over control to the higher layers of the system software. Well actually, the process of booting up the operating system is a bit more involved. The image of the operating system is in the mass storage device. In fact, the BIOS does not even know what operating system needs to be booted up. Therefore, the BIOS has to have a clear idea as to where and how the information is kept in the mass storage device so that it can read in the operating system and hand over control to it. In other words, the layout of the information on the mass storage device becomes a *contract* between the BIOS and the operating system, be it Windows or Linux or whatever else.

To make this discussion concrete, let us assume that the mass storage device is a disk. At the very beginning of the storage space on the disk, say *{platter 0, track 0, sector 0}*, is a special record called the *Master Boot Record (MBR).* Recall that when you create an executable through the compilation process it lives on the disk until loaded into memory by the loader. In the same manner, MBR is just a program at this well-defined location on the disk. BIOS is serving as a loader to load this program into memory and it will transfer control to MBR.

The MBR program knows the layout of the rest of the disk and knows the exact location of the operating system on the disk. The physical disk itself may be made up of several partitions. For example, on your desktop or laptop, you may have seen several "drives" with distinct names (Microsoft Windows gives them names such as C, D, etc.). These may be distinct physical drives but they may also be logical drives, each of which corresponds to a distinct partition on the same physical drive. Let us say you have a single disk but have a dual boot capability that allows either Linux or Windows to be your operating system depending on your choice. The file systems for each of these operating systems necessarily have to be distinct. This is where the partitions come into play.

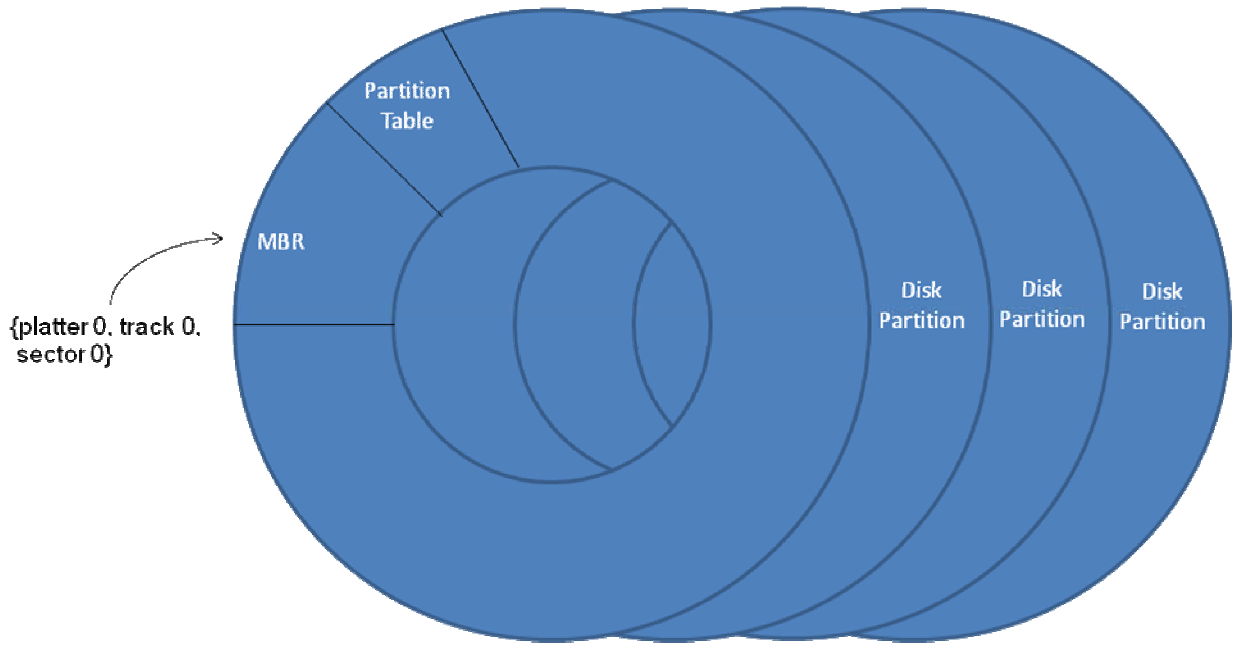


Figure 11.17: Conceptual layout of information on the disk

Figure 11.17 shows a conceptual layout of the disk. For the sake of clarity, we have shown each partition on a different disk platter. However, it should be emphasized that several partitions may live on the same platter depending on the disk capacity. The key data structure of the MBR program is the partition table. This table gives the start and end device address (i.e., the triple *{platter, track, sector}*) of each partition on the disk (Table 11.4). MBR uses this table to decide which partition has to be activated depending on the choice exercised by the user at boot time. Of course, in some systems there may be no choice (e.g., there is a single partition or only one partition has an operating system associated with it).

Partition	Start address {platter, track, sector}	End address {platter, track, sector}	OS
1	{1, 10, 0}	{1, 600, 0}	Linux
2	{1, 601, 0}	{1, 2000, 0}	MS Vista
3	{1, 2001, 0}	{1, 5000, 0}	None
4	{2, 10, 0}	{2, 2000, 0}	None
5	{2, 2001, 0}	{2, 3000, 0}	None

Table 11.4: Partition Table Data Structure

Table 11.4 shows several partitions. Depending on the OS to be booted up, the MBR program will activate the appropriate partition (in this case either partition 1 or 2). Note that partitions 3-5 do not have any OS associated with it. They may simply be logical "drives" for one or the other operating system.

Figure 11.18 shows the layout of information in each partition. Other than the very first entry (boot block), the actual layout of information in the partition varies from file system to file system. However, to keep the discussion concrete, we will assume a particular layout and describe the functionality of each entry. The chosen layout of information is close to traditional Unix file systems.

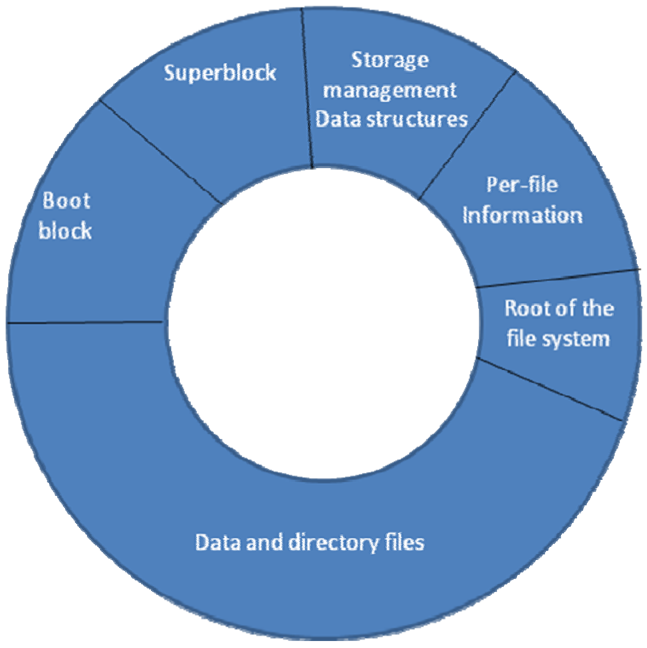


Figure 11.18: Layout of each partition

Let us review each entry in the partition.

- *Boot* block is the very first item in every partition. MBR reads in the boot block of the partition being activated. The boot block is simply a program (just like MBR) that is responsible for loading in the operating system associated with this partition. For uniformity, every partition starts with a boot block even if there is no OS associated with a particular partition (entries 3-5 in Table 11.4).
- *Superblock* contains all the information pertinent to the file system contained in this partition. This is the key to understanding the layout of the rest of this partition. The boot program (in addition to loading the operating system or in lieu thereof) reads in the superblock into memory. Typically, it contains a *code* usually referred to as the *magic number* that identifies the type of file system housed in this partition, the number of disk blocks, and other administrative information pertaining to the file system.
- The next entry in the partition contains the *data structures for storage management* in this partition. The data structures will be commensurate with the specific allocation strategy being employed by the file system (see Section 11.2). For example, it may contain a bit map to represent all the available disk data blocks (i.e., the free list). As another example, this entry may contain the FAT data structure that we mentioned earlier (see Section 11.2.4).
- The next entry in the partition corresponds to the *per-file information* maintained by the file system. This data structure is unique to the specifics of the file system. For example, as we mentioned earlier (see Section 11.3.1), in Unix file system, every file has a unique number associated with it called the i-node number. In such a system, this entry may simply be a collection of all the i-nodes in the file system.
- Modern day file systems are all hierarchical in nature. The next entry in the partition points to the *root directory* of the hierarchical tree-structured file system. For example, in Unix this would correspond to the information pertaining to the file named "/".
- The last entry in the partition is just the collection of disk blocks that serve as containers for *data and directory files*. The data structures in the storage management entry of the partition are responsible for allocating and de-allocating these disk blocks. These disk blocks may be used to hold data (e.g., a JPEG image) or a directory containing other files (i.e., data files and/or other directories).

As should be evident, the superblock is a critical data structure of the file system. If it is corrupted for some reason then it is very difficult to recover the contents of the file system.

11.6.1 In memory data structures

For efficiency, a file system would read in the critical data structures (the superblock, free list of i-nodes, and free list of data blocks) from the disk at the time of booting. The file system manipulates these in-memory data structures as user programs create and delete files. In fact, the file system does not even write the data files created to the mass storage device immediately. The reason for the procrastination is two-fold. First, many files, especially in a program development environment, have a short lifetime (less than 30 seconds). Consider the lifetime of intermediate files created during a program compilation and linking process. The programs (such as the compiler and linker) that create such files will delete them upon creation of the executable file. Thus, waiting for a while to write to the mass storage device is advantageous since such procrastination would help reduce the amount of I/O traffic. The second reason is simply a matter of convenience and efficiency. The file system may batch the I/O requests to reduce both the overhead of making individual requests as well as for dealing with interrupts from the device signaling I/O completion.

There is a downside to this tactic, however. The in-memory data structures may be inconsistent with their counterparts on the disk. One may have noticed the message on the computer screen that says, “It is not safe to remove the device,” when one tries to unplug a memory stick from the computer. This is because the OS has not committed the changes it has made to the in-memory data structures (and perhaps even the data itself) to the mass storage device. Many operating systems offer commands to help you force the issue. For example, the Unix command *sync* forces a write of all the in-memory buffers to the mass storage device.

11.7 Dealing with System Crashes

Let us understand what exactly is meant by an operating system crash. As we have said often, an operating system is simply a program. It is prone to bugs just as any other piece of software that you may write. A user program may be terminated by the operating system if it does something it should not do (try to access a portion of memory beyond its bounds), or may simply hang because it is expecting some event that may never occur. An operating system may have such bugs as well and could terminate abruptly. In addition, a power failure may force an abnormal termination of the operating system. This is what is referred to as a *system crash*.

The file system is a crucial component of the computer system. Since it houses persistent data, the health of the file system is paramount to productivity in any enterprise. Therefore, it is very important that the file system survive machine crashes. Operating systems take tremendous care to keep the integrity of the file system. If a system crashes unexpectedly, there may be inconsistencies between the in-memory data structures at the time of the crash and on-disk versions.

For this reason, as a last ditch effort, the operating system will dump the contents of the memory on to the mass storage device at a well-known location before terminating (be it due to a bug in the OS or due to power failure). On booting the system, one of the first things that the boot program will do is to see if such a crash image exists. If so, it will try to reconstruct the in-memory data structures and reconcile them with the on-disk versions. One of the reasons why your desktop takes time when you boot up your machine is due to the consistency check that the operating system performs to ensure file system integrity. In UNIX, the operating system automatically runs *file system consistency check (fsck)* on boot up. Only if the system passes the consistency check the boot process continues. Any enterprise does periodic backup of the disk on to tapes to ward off against failures.

11.8 File systems for other physical media

Thus far, we have assumed the physical media to be the disk. We know that a file system may be hosted on a variety of physical media. Let us understand to what extent some of the discussion we have had thus far changes if the physical media for the mass storage is different. A file system for a CD-ROM and CD-R (a recordable CD) are perhaps the simplest. Once recorded, the files can never be erased in such media, which significantly reduces the complexity of the file system. For example, in the former case, there is no question of a free list of space on the CD; while in the latter all the free space is at the end of the CD where the media may be appended with new files. File system for a CD-RW (rewritable CD) is a tad more complex in that the space of the deleted files need to be added to the free space on the media. File systems for DVD are similar.

As we mentioned in Chapter 10, solid-state drives (SSD) are competing with disk as mass storage media. Since SSD allows random access, seek time to disk blocks – a primary concern in disk-based file systems – is less of a concern in SSD-based file systems. Therefore, there is an opportunity to simplify the allocation strategies. However, there are considerations specific to SSD while implementing a file system. For example, a given area of an SSD (usually referred to as a *block*) may be written to only a finite number of times before it becomes unusable. This is due to the nature of the SSD technology. Consequently, file systems for SSD adopt a *wear leveling* allocation policy to ensure that all the areas of the storage are equally used over the lifetime of the SSD.

11.9 A summary of modern file systems

In this section, we will review some of the exciting new developments in the evolution of modern file systems. In Chapter 6, we gave a historical perspective of the evolution of Unix paving the way for both Mac OS X and Linux. Today, of course, OS X, Linux and Microsoft Windows rule the marketplace for a variety of application domains. We limit our discussion of modern file systems to Linux and Microsoft families of operating systems.

11.9.1 Linux

The file system interface (i.e., API) provided by Linux has not changed from the days of early Unix systems. However, internally the implementation of the file system has been undergoing enormous changes. This is akin to the architecture vs. implementation dichotomy that we discussed in Chapters 2 and 3 in the context of processor design.

Most of the evolutionary changes are to accommodate multiple file system partitions, longer file names, larger files, and hide the distinction between files present on local media Vs. the network.

One important change in the evolution of Unix file systems is the introduction of *Virtual File System (VFS)*. This abstraction allows multiple, potentially different, file systems to co-exist under the covers. The file system may be on a local device, an external device accessible through the network, and on different media types. VFS does not impact you as a user. You would open, read, or write a file just the same way as you would in a traditional Unix file system. Abstractions in VFS know exactly which file system is

responsible for your file and through one level of indirection (basically, function pointers stored in VFS abstraction layer) redirect your call to the specific file system that can service your request.
Now let us take a quick look at the evolution of file system implementation in Linux itself. In Chapter 6, we mentioned how MINIX served as a starting point for Linux. The very first file system for Linux used the MINIX file system. It had its limitation in the length of file names and maximum size of individual files. So this was quickly replaced by *ext* (which stands for extended file system) that circumvented those limitations. However, ext had several performance inefficiencies giving rise the *ext2*, a second version of ext, which is still in widespread use in the Linux community.

11.9.1.1 ext2

The disk layout of an ext2 file system partition is not remarkably different from the generic description we gave in Section 11.6. A few things are useful to mention regarding ext2.

- **Directory file**

The first is the contents of an i-node structure for a directory file. A directory file is made up of an integral number of contiguous disk blocks to facilitate writing the directory file to the disk in one I/O operation. Each entry in the directory names a file or another sub-directory. Since a name can be of arbitrary length the size of each entry is variable. Figure 11.19-(a) shows the format of an entry in the directory.

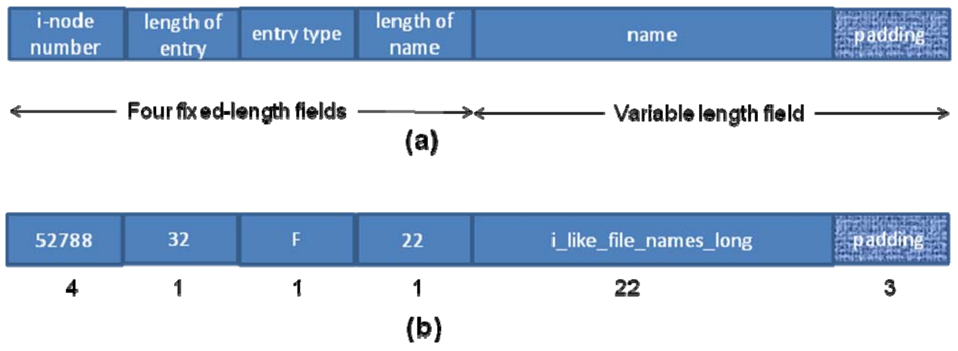


Figure 11.19: Format of each individual entry in a directory file

Each entry is composed of:

- **i-node number**: a fixed length field that gives the i-node number for the associated entry
- **length of entry**: a fixed length field that gives the length of the entry in bytes (i.e., the total amount of space occupied by the entry on the disk); this is needed to know where in the directory structure the next entry begins
- **type of entry**: a fixed length field that specifies if the entry is a data file (f) or a directory file (d)
- **length of name**: a fixed length field that specifies the length of the file name
- **name**: a variable length field that gives the name of the file in ASCII

- **padding**: an optional variable length field perhaps to make the total size of an entry some multiple of a binary power; as we will see shortly, such padding space may also be created or expanded due to file deletions

Figure 11.19-(b) shows an example entry for a file named "i_like_my_file_names_long" with the values filled in. The size of each field in bytes is shown below the figure.

Entries in the directory are laid out contiguously in the textual order of creation of the files. For example, if you create the files "datafile", "another_datafile", and a "my_subdirectory", in that order in a directory; the first two being data files, and the third a directory file. The directory entries will be as shown in Figure 11.20-(a). Let us say, you delete one of the files, say, "another_datafile". In this case, there will simply be an empty space in the directory layout (see Figure 11.20-(b)). Basically, the space occupied by the deleted file entry will become part of the padding of the previous entry. This space can be used for the creation of a new file entry in the directory.

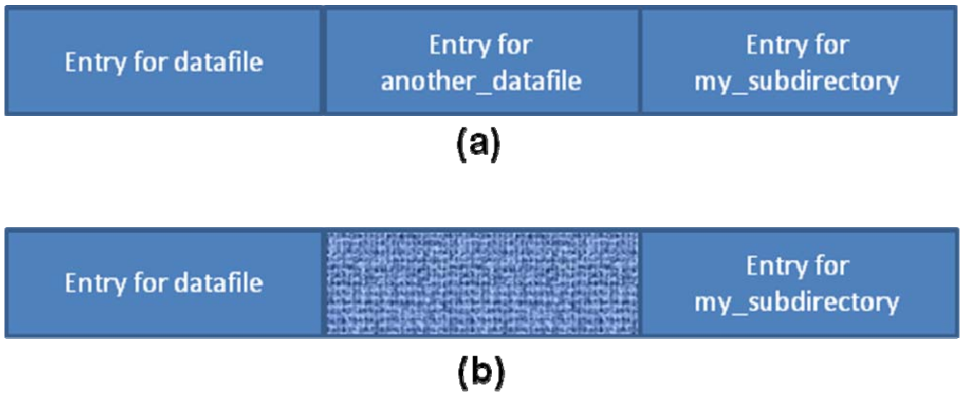

Entry for datafile
Entry for another_datafile
Entry for my_subdirectory
(a)
Entry for datafile
Entry for my_subdirectory
(b)

Figure 11.20: Layout of multiple entries in a directory file

- **Data file**

The second point of interest is the i-node structure for a data file. As we said earlier, ext2 removes the limitation of MINIX with respect to the maximum size of files. The i-node structure for a data file reflects this enhancement. The scheme used is the hybrid allocation scheme we discussed earlier in Section 11.2.7. An i-node for a data file contains the following fields:

- **12 data block addresses**: The first 12 blocks of a data file can be directly reached from the i-node. This is very convenient for small files. The file system will also try to allocate these 12 data blocks contiguously so that the performance can be optimized.
- **1 single indirect pointer**: This points to a disk block that serves as a container for pointers to data blocks. For example, with a disk block size of 512 bytes, and a data block pointer size of 4 bytes, this first level indirection allows expanding the file size by an additional 128 data blocks.
- **1 double indirect pointer**: This points to a disk block that serves as a container for pointers to tables that contain single level indirect pointers. Continuing the same example, this second level indirection allows expanding the file size by an additional 128*128 (i.e., 2^{14}) data blocks.

- **1 triple indirect pointer**: This adds one more level of indirection over the double indirect pointer, allowing a file to be expanded by an additional 128*128*128 (i.e., 2^{21}) data blocks.

Figure 11.21 pictorially shows such an i-node with all the important fields filled in.

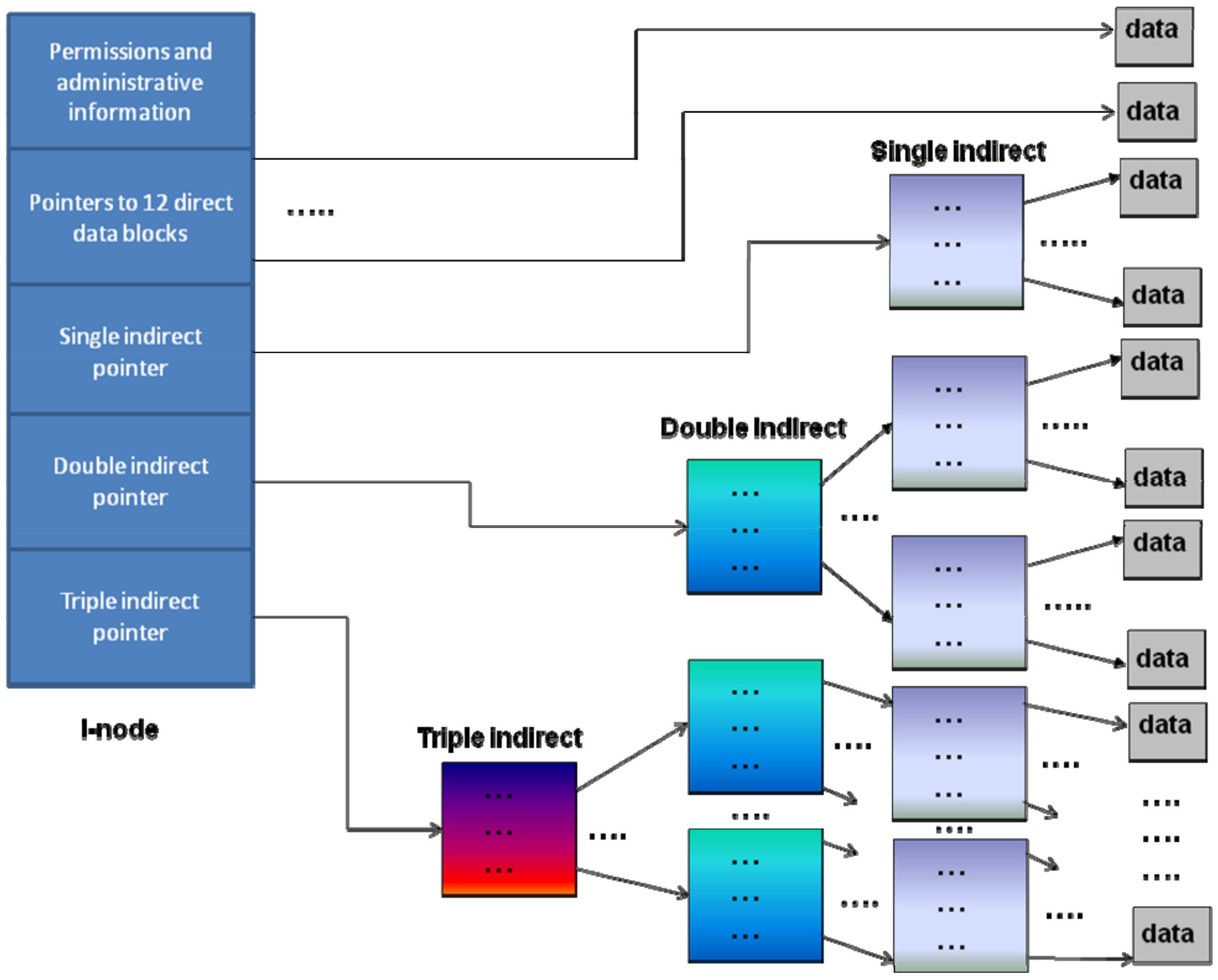


Figure 11.21: i-node structure of a data file in Linux ext2

Example 8:

What is the maximum file size in bytes possible in the ext2 system? Assume a disk block is 1 KB and pointers to disk blocks are 4 bytes.

Answer:

<u>Direct data blocks</u>

Number of direct data blocks	=	number of direct pointers in an i-node	
	=	12	(1)

Data blocks via the single indirect pointer

Next, let us count the number of data blocks available to a file via the single indirect pointer in an i-node. The single indirect pointer points to a single indirect table that contains pointers to data blocks.

Number of data blocks via the single indirect pointer in an i-node
= Number of pointers in a single indirect table
= Number of pointers in a disk block
= size of disk block/size of pointer
= 1KB/4bytes
= 256 (2)

Data blocks via double indirect pointer

Next, let us count the number of data blocks available to a file via the double indirect pointer in an i-node. The double indirect pointer points to a disk block that contains pointers to tables of single indirect pointers.

Number of single indirect tables via the double indirect pointer
= Number of pointers in a disk block
= size of disk block/size of pointer
= 1KB/4bytes
= 256 (3)

Number of data blocks via each of these single indirect tables (from equation (2) above)
= 256 (4)

From (3) and (4), the number of data blocks via the double indirect pointer in an i-node
= 256*256 (5)

Data blocks via triple indirect pointer

By similar analysis the number of data blocks via the triple indirect pointer in an i-node
= 256*256*256 (6)

Putting (1), (2), (5), and (6) together, the total number of disk blocks available to a file
= 12 + 256 + 256*256 + 256*256*256
= $12 + 2^8 + 2^{16} + 2^{24}$

The maximum file size in bytes for a data file (given the data block is of size 1 KB)
= $(12 + 2^8 + 2^{16} + 2^{24})$ KB
> **16 GBytes**

11.9.1.2 Journaling file systems

Most modern file systems for Linux are *journaling* file systems. Normally, you think of writing to a file as writing to the data block of the file. Logically, this is correct. However, practically there are issues with this approach. For example, if the file size is too small then there is both space and more importantly time overhead in writing such small files to the disk. Of course, operating systems buffer writes to files in memory and write to disk opportunistically. Despite such optimizations, ultimately, these files have to be written to disk. Small files not only waste space on the disk (due to internal fragmentation) but also result in time overhead (seek time, rotational latency, and metadata management).

Another complication is system crashes. If the system crashes in the middle of writing to a file, the file system will be left in an inconsistent state. We alluded to this problem already (see Section 11.7).

To address both the problem of overcoming the inefficiency of writing small files and to aid in the recovery from system crashes, modern file systems use a journaling approach. The idea is simple and intuitive and uses the metaphor of keeping a personal journal or a diary. One's personal diary is a time-ordered record of events of importance in one's everyday activities. The *journal* serves the same purpose for the file system. Upon file writes, instead of writing to the file and/or creating a small file, a log record (similar to database record) that would contain the information corresponding to the file write (meta-data modifications to i-node and superblock, as well as modifications to the data blocks of the file). Thus, the journal is a *time-ordered* record of all the changes to the file system. The nice thing about this approach is that the journal is distinct from the file system itself, and allows the operating system to optimize its implementation to best suit the file system needs. For example, the journal may be implemented as a sequential data structure. Every entry in this sequential data structure represents a particular file system operation.

For example, let us say you modified specific data blocks three files (X, Y, and Z) that are scattered all over the disk in that order. The journal entries corresponding to these changes will appear as shown in Figure 11.22. Note that each log record may of different size depending on the number of data blocks of a file that is modified at a time by the corresponding write operation.

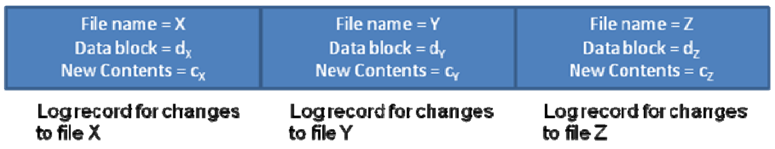


Figure 11.22: Journal entries of changes to the file system

The journal data structure is a composite of several *log segments*. Each log segment is of finite size (say for example, 1 MByte). As soon as a log segment fills up, the file system writes this out to a contiguous portion of the disk and starts a new log segment for the subsequent writes to the file system. Note that the file X, Y, and Z do not reflect the changes made to them yet. Every once in a while, the file system may *commit* the changes to the actual files by reading the log segment (in the order in which they were generated) and applying the log entries to the corresponding files. Once the changes have been committed then the log segments may be discarded. If the file X is read before the changes have been applied to it, the file system is smart enough to apply the changes from the log segment to the file before allowing the read to proceed.

As should be evident, the log segment serves to aggregate small writes (to either small files, or small parts of a large file) into coarser-grain operations (i.e., larger writes to contiguous portion of the disk) to the journal.

Journaling overcomes the small write problem. As a bonus journaling also helps coping with system crashes. Remember that as a last ditch effort (see Section 11.7), the operating system stores everything in memory to the disk at the time of the crash or power failure. Upon restart, the operating system will recover the in-memory log segments. The file system will recognize that the changes in the log segments (both the ones on the disk as well as the in-memory log segments that were recovered from the crash) were not committed successfully. It will simply reapply the log records to the file system to make the file system consistent.

Ext3 is the next iteration of the Linux file system. Compared the ext2, the main extension in the ext3 file system is support for journaling. In this sense, a file partition built using ext2 may be accessed using ext3 file system since the data structures and internal abstractions are identical in both. Since creating a journal of all file system operations may be expensive in terms of space and time, ext3 may be configured to journal only the changes to the meta-date (i.e., i-nodes, superblock, etc.). This optimization will lead to better performance in the absence of system crashes but cannot make guarantees about file data corruption due to crashes.

There are a number of new Unix file systems. *ReiserFS* is another file system with meta-data only journaling for Linux. *jFS* is a journaling file system from IBM for use with IBM's AIX (a version of Unix) and with Linux. *xFS* is a journaling file system primarily designed for SGI Irix operating system. zFS is a high performance file system for high-end enterprise systems built for the Sun Solaris operating system. Discussion of such file systems is beyond the scope of this textbook.

11.9.2 Microsoft Windows

Microsoft file systems have an interesting history. As one may recall, MS-DOS started out as an operating system for PC. Since disks in the early stages of PC evolution had fairly small capacity (on the order of 20-40 MBytes), the file systems for such computers were limited as well. For example, FAT-16 (a specific instance of the allocation scheme discussed in Section 11.2.4) uses 16-bit disk addresses, with a limit of 2Gbytes per file partition. FAT-32, by virtue of using 32-bit disk addresses, extends the limit of a file partition to 2 TBytes. These two file systems have mostly been phased out with the advent of Microsoft XP and Vista operating systems. Currently (circa 2008), NT file system, which was first introduced with the Microsoft NT operating system, is the *de facto* standard Microsoft file system. FAT-16 may still be in use for removable media such as floppy disk. FAT-32 also still finds use, especially for inter-operability with older versions of Windows such as 95/98.

NTFS, as the file system is referred to, supports most of the features that we discussed in the context of Unix file systems earlier. It is a complex modern file system using 64-bit disk address, and hence can support very large disk partitions. *Volume* is the fundamental unit of structuring the file system. A volume may occupy a part of the disk, the whole disk, or even multiple disks.

API and system features. A fundamental difference between NTFS and Unix is the view of a file. In NTFS, a file is an *object* composed of *typed attributes* rather than a stream of bytes in Unix. This view of a file offers some significant flexibility at the user level. Each typed attribute is an independent byte stream. It is possible to create, read, write, and/or delete each attributed part without affecting the other parts of the same file. Some attribute types are standard for all files (e.g., name, creation time, and access control). As an example, you may have an image file with multiple attributes: raw image, thumbnail, etc. Attributes may be created and deleted at will for a file.

NTFS supports long file names (up to 255 characters in length) using Unicode character encoding to allow non-English file names. It is a hierarchical file system similar to Unix, although the hierarchy separator in Unix, namely, "/", is replaced by "\" in NTFS. Aliasing a file through hard and soft links, which we discussed in Section 11.1, is a fairly recent addition to the NTFS file system.

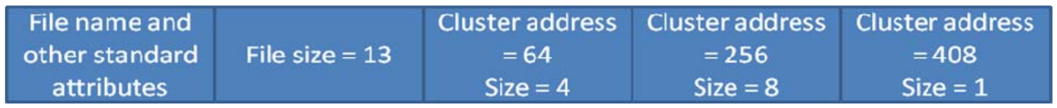
Some of the features of NTFS that are interesting include on the fly compression and decompression of files as they are written and read, respectively; an optional encryption feature; and support for small writes and system crashes via journaling.

Implementation. Similar to the i-node table in Unix (see Section 11.3.1), the main data structure in NTFS is the *Master File Table (MFT)*. This is also stored on the disk and contains important meta-data for the rest of the file system. A file is represented by one or more records in the MFT depending on both the number of attributes as well as the size of the file. A bit map specifies which MFT records are free. The collection of MFT records describing a file is similar in functionality to an i-node in the Unix world but the similarity stops there. An MFT record (or records) for a file contains the following attributes:

- File name
- Timestamp
- Security information (for access control to the file)
- Data or pointers to disk blocks containing data
- An optional pointer to other MFT records if the file size is too large to be contained in one record or if the file has multiple attributes all of which do not fit in one MFT record

Each file in NTFS has a unique ID called an *Object reference*, a 64-bit quantity. The ID is an index into the MFT for this file and serves a similar function to the i-node number in Unix.

The storage allocation scheme tries to allocate contiguous disk blocks to the data blocks of the file to the extent possible. For this reason, the storage allocation scheme maintains available disk blocks in *clusters*, where each cluster represents a number of contiguous disk blocks. The cluster size is a parameter chosen at the time of formatting the drive. Of course, it may not always be possible to allocate all the data blocks of a file to contiguous disk blocks. For example, consider a file that has 13 data blocks. Let us say that the cluster size is 4 blocks, and the free list contains clusters starting at disk block addresses 64, 256, 260, 408. In this case, there will be 3 non-contiguous allocations made for this file as shown in Figure 11.23.



File name and other standard attributes	File size = 13	Cluster address = 64 Size = 4	Cluster address = 256 Size = 8	Cluster address = 408 Size = 1

Figure 11.23: An MFT record for a file with 13 data blocks

Note that internal fragmentation is possible with this allocation since the remaining 3 disk blocks in cluster 408 are unused.

An MFT record is usually 1Kbytes to 4 Kbytes. Thus, if the file size is small then the data for the file is contained in the MFT record itself, thereby solving the small write problem. Further, the clustered disk allocation ensures good sequential access to files.

11.10 Summary

In this chapter, we studied perhaps one of the most important components of your system, namely, the file system. The coverage included

- attributes associated with a file,
- allocation strategies and associated data structure for storage management on the disk,
- meta-data managed by the file system,
- implementation details of a file system,
- interaction among the various subsystems of the operating system,
- layout of the files on the disk,
- data structures of the file system and their efficient management,
- dealing with system crashes, and
- file system for other physical media

We also studied some modern file systems in use circa 2008.

File systems is a fertile of research and development. One can see an analogy similar to processor architecture vs. implementation in the context of file systems. While the file system API remains mostly unchanged across versions of operating systems (be they Unix, Microsoft, or anything else), the implementation of the file system continually evolves. Some of the changes are due to the evolution in the disk technology, and some are due to the changes in the types of files used in applications. For example, modern workloads generate multimedia files (audio, graphics, photos, video, etc.) and the file system implementation has to adapt to supporting such diverse file types efficiently.

In this chapter, we have only scratched the surface of the intricacies involved in implementing a file system. We hope that we have stimulated your interest enough to get you to take a more advanced course in operating systems.

11.11 Review Questions

1. Where can the attribute data for a file be stored? Discuss the pros and cons of each choice.

2. Examine the following directory entry:

```
-rwxrwxrwx   3 rama          0 Apr 27 21:01 foo
```

   After executing the following commands:

```
chmod u-w foo
chmod g-w foo
```

   What are the access rights of the file "foo"?

3. Linked allocation results in
   1. Good sequential access
   2. Good random access
   3. Ability to grow the file easily
   4. Poor disk utilization
   5. Good disk utilization

   Select all that apply

4. Fixed contiguous allocation of disk space results in

   1. Good sequential access
   2. Good random access
   3. Ability to grow the file easily
   4. Poor disk utilization
   5. Good disk utilization

   Select all that apply.

5. What is the difference between hard and soft links?

6. For ext2 file system with a disk block size of 8 KB and a pointer to disk blocks of 4 bytes, what is the largest file that can be stored on the system? Sketch the structure of an i-node and show the calculations to arrive at the maximum file size (see Example 8).