chapter

17

File System Organization

17.1 FILE SYSTEM ORGANIZATION

17.1.1 What Is a File System?

So far we have files, open files, and directories as file system objects. These objects are represented by descriptors. But where do these objects reside? It is convenient to have an overall object that holds them, and we will call this object a file system. The file system object will have its own descriptor. A **file system** is a collection of files, blocks, directories, and file descriptors, all on one logical disk.[1]

There is a possible confusion here between the general term "file system" for the part of the operating system that implements files, and this "file system" that is a collection of files and directories. We will try to be clear on which is meant if there might be a confusion.

A file system is the largest unit of structure placed on disks.

17.1.2 File System Structure

A file system is stored on a logical disk. As we saw in Chapter 15, this might be a physical disk, part of a physical disk, or several physical disks. A disk is essentially a large array of disk blocks, so we have to lay out the parts of the file system on this array of disk blocks. Figure 17.1 shows an example file system layout. We will examine this layout first, and then think about the variations that are possible.

[1] Open files are not part of the "file system" that is the data structure on disk, but they are part of the "file system" that is the part of the operating system that implements files.

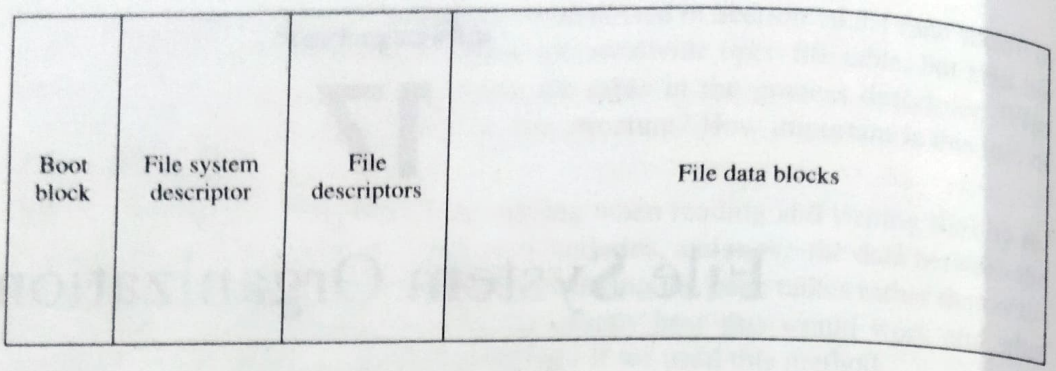


Figure 17.1 Layout of a file system (not to scale)

Usually, the first block of a disk is reserved as a boot block. We will talk more about this in Section 17.7. For now, we will just reserve block 0 for that purpose, and start the file system in block 1.

The entire file system is an object itself and needs a data structure to represent it. So we will reserve block 1 for the file system descriptor. We will examine the contents of the file system descriptor in Section 17.1.3.

It is common to keep the file descriptors in a special place on the disk, so we will allocate the next part of the disk for file descriptors. File descriptors are generally smaller than disk blocks, so several file descriptors will fit into each disk block in this section. Let us suppose that eight of them fit into each block, and we allocate 1000 blocks for this purpose. This gives us a maximum of 8000 files on this disk.

Since the file descriptors are all in one place, we can address them with numbers. File descriptor 0 will be the first one in block 2, file descriptor 1 will be the second one in block 2, file descriptor 8 will be the first one in block 3 and so on. This gives us nice short addresses for file descriptors. Alternatively, we could use a pair <disk block address, fd number (0 to 7)> to address file descriptors. The address of the file descriptor is the internal name of the file. We will examine file descriptors in more detail in Section 17.2.

The rest of the disk is allocated to blocks that will hold all the file data. Directories are implemented as files, so each directory will have a file descriptor in the file descriptor area and some data blocks in the data block area. There is not a special area in the file system for directories.

Free Blocks Some of the data blocks will be allocated and some will be free. All the allocated blocks will be linked to a file descriptor, so we could find a free block by following all the pointers in the file descriptors, finding all the allocated blocks, and assuming that all the rest of the blocks are free. This is, in fact, what we do if we are reconstructing a damaged disk, but for normal operation we will want to keep a list of free blocks for easy and fast allocation of blocks.

All disk blocks are the same, and so we just keep them in a single list. Typically, the list will be kept in some of the free blocks, themselves. Each block contains a number of addresses of free blocks, and the last address is the address of the free block that contains the next block of free block addresses.

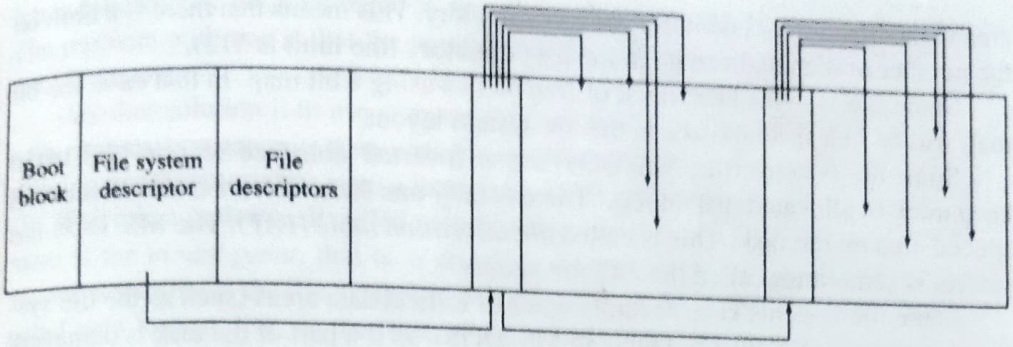


Figure 17.2 Free list organization

Figure 17.2 shows how the free list would be stored in a file system. The head of the free block list is in the file descriptor.

17.1.3 The File System Descriptor

Now we are in a position to define the contents of the *file system descriptor*:

file system descriptor

- The total size of the file system (in blocks). This is the size of the logical disk the file system is stored on.
- The size of the file descriptor area.
- The first block on the free block list.
- The location of the file descriptor of the root directory.
- The time the file system was created, last modified, and last used.
- Other file system meta-data (e.g., if the file system is read-only).

We need the information about how the rest of the file system is laid out so we can find the file descriptors and the data block area. We need to find the beginning of the free list and the root directory. Often there is other information kept about the file system. For example, sometimes we have read-only file systems.

In UNIX, file system descriptors are called *superblocks*.

superblock

17.1.4 Variations in File System Layout

file system layout

We might decide to keep the file descriptors in ordinary data blocks, instead of in a special place in the file system. The problem here is how to name them. If we only have one file descriptor per block, we can name it with the number of the disk block it is in. But then we will use an entire disk block for each file descriptor. Alternatively we could pack several file descriptors into one disk block and give it a two-part name: block number, and file descriptor number within that block.

In UNIX, the first file descriptor is always the root directory. This avoids having to record that information in the file system descriptor. MS/DOS allocates a special

area in the file system layout for the root directory. This means that there is a limit on the number of files that can go in the root directory (the limit is 512).

Some file systems keep track of free blocks using a bit map. In that case, the bit map will be in a special place in the file system layout.

Some file systems (like MS/DOS) use an inverted table (see Section 17.3.14) to keep track of allocated disk blocks. There is only one such table, and it is allocated a special area on the disk. This is called *file allocation table (FAT)*. The MS/DOS file system is sometimes called the *FAT file system*.

Some file systems keep several copies of critical data areas (such as the file system descriptor or the inverted table in MS/DOS), so if a part of the disk is damaged, the rest can still be recovered.

If the file descriptors are at one end of the disk, they might be far away from the file data blocks they point to, and this leads to long seeks. The Berkeley Fast File System (FFS) divides the disk into a number of "cylinder groups." Each cylinder group has a copy of the file system descriptor (called the superblock in UNIX), a file descriptor space, and a data block space. This way the directories, file descriptors, and the file are close together on the disk. In addition, this provides multiple copies of the superblock in case the master copy is damaged.

17.1.5 File Systems in Disk Partitions

We have been talking about the file system being on a disk, but all we have assumed is that the disk is a large array of disk blocks. Often a file system is in a partition of a disk, and not a whole disk. Or we could use a device driver that combined several disks into a large logical disk, and put a large file system on the logical disk.

Sometimes different types of file systems will be in the different partitions of a disk. This allows you to run more than one operating system (with differently structured file systems) from the same physical disk.

Also, some operating systems support several types of file system. For example, one partition might hold an ordinary file system, and another partition might hold a mirroring file system that makes two copies of all data and provides higher reliability.

17.1.6 Combining File Systems

Modern computers do not have just one disk and often have dozens of disks, so it is necessary to have files on several disks. The file system layout we described is intended to exist on a single disk. This is a logical disk, so it can be part of a disk or several disks. However, it is not convenient to combine all your disks into a single large logical disk. It makes backups (see Section 17.9.1) harder to do. It makes it harder to add disks and remove disks, and a failure in any one disk will bring down the entire file system. So we need a way to combine file systems into a single name space while keeping each file system separate from the others.

The first issue is the naming system. A simple solution taken by MS/DOS is to give each disk a letter name, so there are disks 'A,' 'B,' 'C,' etc. A file name is a disk

letter followed by the path name in that file system, for example `C:\usr\bin\ls`. The problem with this is that the naming system is nonuniform: we have letters and then file names.

Another solution it to use a concept called *mounting*. If you have a directory in a file system, you can mount another file system on that directory. You are essentially splicing one directory tree onto a branch in another directory tree. mounting

There is a system call called *mount* which takes two arguments. The first argument is the *mount point*, that is, a directory in the current file naming system. The second argument is the file system to mount at that point. This file system will be located on a logical disk, so the form of the second argument will be the name of the logical disk the file system is on.

When we get to a mounted-on directory in a path name search, we jump to the root directory of the mounted file system and continue the search from there. Figure 17.3

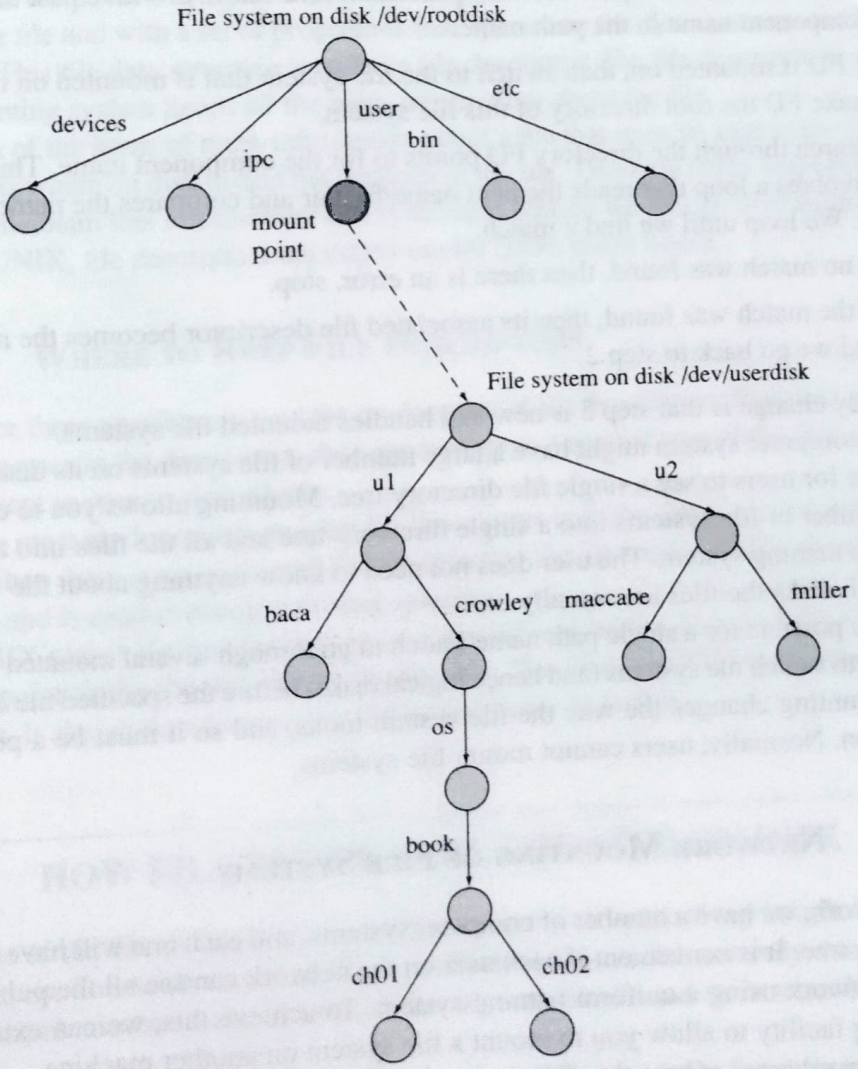


Figure 17.3 Mounting a file system

shows how this would work. The file system on the disk `/dev/rootdisk` is the root file system, and it is the only file system accessible when the operating system starts up. During the system initialization process, the user file system on device `/dev/userdisk` will be mounted on the `/user` directory in the root file system. The user tree is grafted on to the root tree at `/user`. This combines the two directories into a single directory.

The path name search algorithm has to be modified slightly to handle mounted file systems. Here is the new algorithm for looking up a path name.

1. If the path name starts with the '/' then let FD be the root directory and move the name pointer past the '/'. Otherwise let FD be the current working directory, and start at the beginning of the path name.
2. If we are at the end of path name, then return FD and stop.
3. If FD is not a directory, then there is an error, stop.
4. Isolate the next component in the path name and call it C. Move past the component name in the path name.
5. If FD is mounted on, then switch to the file system that is mounted on FD and make FD the root directory of this file system.
6. Search through the directory FD points to for the component name. This involves a loop that reads the next name/fd pair and compares the name with C. We loop until we find a match.
7. If no match was found, then there is an error, stop.
8. If the match was found, then its associated file descriptor becomes the new FD, and we go back to step 2.

The only change is that step 5 is new and handles mounted file systems.

A computer system might have a large number of file systems on its disks, but it is easier for users to see a single file directory tree. Mounting allows you to combine any number of file systems into a single directory tree and all the files into a single, uniform naming system. The user does not need to know anything about file systems or which disks the files are actually on.

It is possible for a single path name search to go through several mounted directories and to switch file systems (and hence logical disks) before the specified file is found.

Mounting changes the way the file system looks, and so it must be a protected operation. Normally, users cannot mount file systems.

17.1.7 Network Mounting of File Systems

In a network, we have a number of computer systems, and each one will have its own directory tree. It is convenient if each user on the network can see all the public files in the network using a uniform naming system. To achieve this, we can extend the mounting facility to allow you to mount a file system on another machine.

The mechanics of how this is done involve many details, but the basic idea is just the same as for a single file system. When a path name search reaches a mounted file

system, it moves to the root directory on the mounted file system. If that file system is on another machine, then it will have to communicate with the other machine in order to complete the path name search. See Section 17.11.1 about the vnode/vfs architecture which supports network mounting.

Mounting allows you to logically combine the naming systems of a collection of file systems.

17.2 FILE DESCRIPTORS

A file is an abstract object implemented by the operating system, and we will implement it in the usual way, with a data structure that records the information we keep about the file and with a set of procedures that operate on the data structure and hence the file. This file data structure is called a file descriptor. The file descriptor is where the operating system keeps all the meta-information about the file.

One of the items of meta-information about a file that must be kept in the file descriptor is where on disk to find the contents of the file. There are many ways in which we can maintain this information, and in the next section we will go over them.

In UNIX, file descriptors are called *inodes* (from index node).

inode

17.2.1 WHERE TO KEEP FILE DESCRIPTORS

There are three possible choices for the location of file descriptors. They can go with the file names in the directories, they can be kept in a special area of the disk, or they can be kept in regular disk blocks.

The most obvious place to keep file descriptors is in the directory along with the name. Then there is no additional disk read to find the descriptor. This is the simplest method and is used in many operating systems.

UNIX places the file descriptors in a special area on the disk, and the directory entries point to the file descriptors in this area. This makes UNIX links easy to implement. It also makes directories smaller and faster to search.

17.3 HOW FILE BLOCKS ARE LOCATED ON DISK

The user of the file abstraction sees a file as an array of bytes of any length. We need to figure out how to store such a file on a physical disk. A disk can only store data in fixed-size blocks, so we have to break the file into blocks and store these blocks on the disk.

Figure 17.4(a) shows a logical file that is 6,658 bytes long. Suppose we have a disk that reads and writes in blocks 1K (1024 bytes) long. The first step in storing the

file on disk is to divide it into pieces that are one block long. Figure 17.4(b) shows the same file divided into seven 1K blocks. The last block is only partially filled, but the file descriptor will keep track of how long the file really is, and the extra bytes will be ignored by the operating system. Each of these disk blocks will be stored somewhere on the disk, and we need to be able to find all the blocks in a file.

logical blocks
logical block numbers
physical blocks
physical block numbers

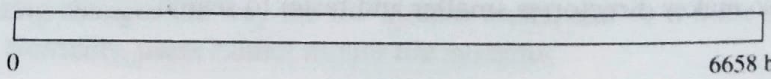
These blocks will be called the *logical blocks* of the file. They are numbered from 0 with *logical block numbers*. We shall see that these blocks will be stored on the disk in *physical blocks* with *physical block numbers*. When we get a request to read a specific byte in the file, we are given its logical byte number in the file. We convert this to the logical block number it is in (by dividing it by the block size), and then we have to find the physical block number of the physical block that is holding this logical block.

17.3.1 The Block Mapping Problem

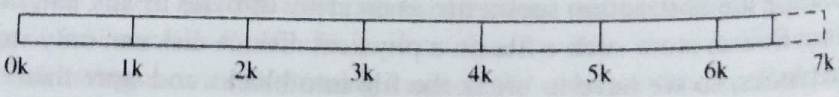
Here is the problem we need to solve. When we create a file, we need to allocate space for it on the disk, and we need to keep track of the space. When a file is created, the creating process might not have any idea how big the file will eventually get. For example, most operating systems keep various "log" files that record system activities, such as all logins, all commands, all file accesses, etc. These files keep growing steadily as events occur. We would like to be able to create a file without giving a maximum length, and just have the file grow as we write data into it.

When we read from the file, we give a logical byte offset to start reading from and a length. We need to be able to convert a logical block number to a physical block number. So we have to decide:

- How to store logical file blocks on the disk.
- How to allocate space to files as they grow.
- How to find a logical file block on the disk.
- How to record the necessary information in the file descriptor and other disk blocks.



(a) A logical file (6658 bytes long)



(b) File divided into 1k blocks

Figure 17.4 File divided into blocks

17.3.2 Contiguous Files

The simplest solution is to keep the files contiguously on disk. The file descriptor only needs to keep the address of the first disk block, and the file length will tell us how many disk blocks it uses. Figure 17.5 shows a *contiguous file*.

contiguous file

Contiguous allocation makes it very efficient to read (and write) in the file. Disks can read physically contiguous sectors at maximum disk speed. Contiguous organization is very good for data that must be read or written at a high rate of speed, such as video data.

This speed comes at a high price in terms of disk space management, however. Contiguous files on disk are even more difficult to deal with than contiguous blocks of memory. One reason is that disks are much slower than memory, and rearranging files on a disk is a very slow operation indeed. There are two problems. The first is fragmentation. Often it will be hard to find enough contiguous space to put a file. Since we really cannot make a process wait for a file to be freed and space to become available, we would have to rearrange the existing files on the disk to coalesce the empty space into one large enough.

File systems have another problem, and that is that we usually do not know how long a file is going to be when it is created. The user creates the file and writes that data to it a little at a time. We could make the user state ahead of time how much space the file will require, and, in fact, some early operating systems did do that. But this is quite an inconvenience for the users because often they will not know how long a file is going to be. Often files are extended even after they have been created and closed for the first time. In these cases, we would have to make extra space at the end of the file, or else move the file to a bigger hole. For example, in Figure 17.6 we can extend File1 and File4 without moving them, but to extend File2 or File3 we would have to move the file or the one next to it.

These problems are serious enough that contiguous allocation of file blocks is only used in special cases where maximum speed is required. It is not really a feasible method for a general file system.

Note that this is similar to the decisions we made in memory management when we decided not to store a process's memory in physically contiguous memory, but to

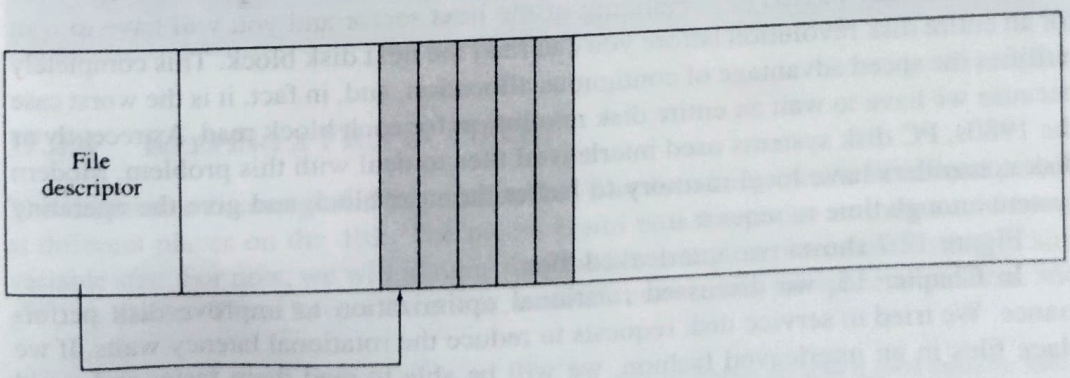


Figure 17.5 A contiguous file

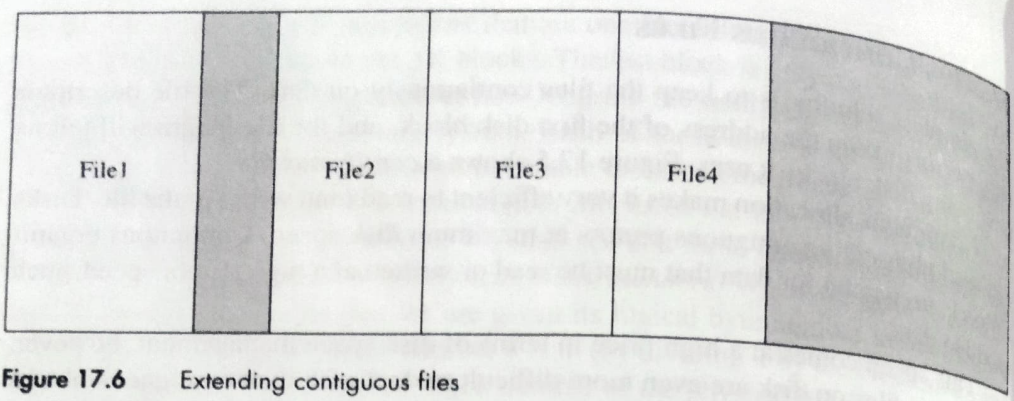


Figure 17.6 Extending contiguous files

divide it up into several pieces. This made the management of memory space much easier, just as it makes disk space management much easier.

Also, even if we allow files to be stored in separate pieces, there will be no rule against keeping them physically together for those files that require this. There are programs called *disk compactors* that do exactly this. We'll talk more about this in Section 17.3.17.

Contiguous files are fast to read and write, but difficult to allocate.

17.3.3 Interleaved Files

Instead of keeping files in physically contiguous disk blocks, we can instead keep them in every other disk block or maybe every third disk block. This may seem silly, but in fact there used to be a good reason for storing files this way. The reason was that, with many disk controllers, it is not easy to read two consecutive disk blocks because, by the time you get the disk interrupt that the first one had been read, figure out that you want to read the next one, and issue the next disk request, the disk head will have already passed the beginning of the next sector and you will have to wait for an entire disk revolution before you can read the next disk block. This completely nullifies the speed advantage of contiguous allocation, and, in fact, it is the worst case because we have to wait an entire disk revolution for each block read. As recently as the 1980s, PC disk systems used interleaved files to deal with this problem. Modern disk controllers have local memory to buffer the next block and give the operating system enough time to request it.

Figure 17.7 shows two interleaved files.

In Chapter 15, we discussed rotational optimization to improve disk performance. We tried to service disk requests to reduce the rotational latency waits. If we place files in an interleaved fashion, we will be able to read them faster and avoid extra rotations. We mentioned that many modern disk controllers will read additional

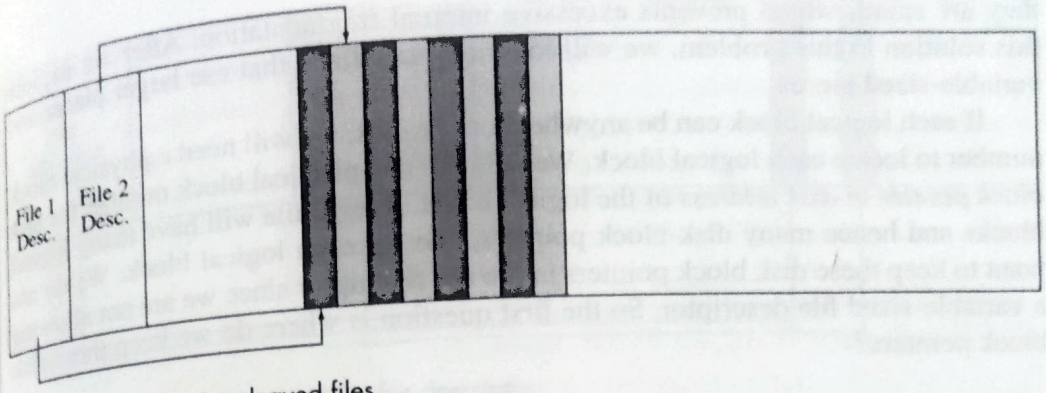


Figure 17.7 Two interleaved files

blocks and buffer them in case they get a request for them. The Berkeley Fast File System (FFS) tries to lay out file blocks so that they can be accessed in the least number of disk rotations. Sometimes it is not possible to store blocks sequentially on a disk. In these cases, the FFS tries to place the block in the next free space rotationally. This might be on a different track in the same cylinder. In this case, we have to interleave because even a smart disk controller cannot know which track to buffer. So they leave in enough rotational time to allow the file system to get the request out to the disk and for the disk to switch read heads.[2]

We bring up the topic of interleaved files here for another reason. There are two separate advantages of contiguous allocation. The first is reading and writing speed, and the second is the fact that you only needed the address of the first block to know the addresses of all the blocks in the file. Interleaved files also share this second advantage. The issue is that, in these cases, there is an algorithm to find the physical block number of logical block $N + 1$ if you know the physical block number of logical block N. We will see in the next section that losing this algorithm means that we have to expend a fair amount of time and space to keep track of where the disk blocks are.

Interleaved files are best when there might be a delay in requesting logically consecutive blocks in a file

17.3.4 Keeping a File in Pieces

The alternative to contiguous (or interleaved) files is to keep the file in several pieces at different places on the disk. The pieces could be a fixed size or they could be of variable size. For now, we will assume that the file is kept in pieces consisting of single disk blocks. These have a fixed size, which is convenient for space allocation, and

[2] Often a disk can switch heads in essentially zero time, but some controllers do have a small delay for head switching.

they are small, which prevents excessive internal fragmentation. After we explore this solution to this problem, we will look into variations that use larger pieces and variable-sized pieces.

isk block pointer
isk address

If each logical block can be anywhere on the disk, we will need a physical block number to locate each logical block. We will call this physical block number the *disk block pointer* or *disk address* of the logical block. A large file will have many logical blocks and hence many disk block pointers, one for each logical block. We do not want to keep these disk block pointers in the file descriptor since we are not allowing a variable-sized file descriptor. So the first question is where do we keep these disk block pointers?

17.3.5 Where to Keep the Disk Block Pointers

There will be one disk block pointer for each disk block. If we assume that disk blocks are 4K long and disk addresses are 4 bytes long, then the list of disk block addresses will be about 0.1 percent as long as the file itself. So a 2 Mbyte file will require 2 Kbytes of file pointers. This presents us with a recursive file storage problem: where do we put all these file pointers on disk? This is recursive because the problem we were trying to solve was where to put the disk blocks of the file on the disk. We have reduced the problem to one that is 1024 times smaller (assuming a 4 Kbyte disk block and 4 byte disk pointers), so we have made some progress.

There are a number of possible solutions which we will go through in the next few sections. We will go through a logical sequence of solutions where the problems in one solution lead us to the next solution. This is what we did with memory management. Most of the solutions will not be practical for real file systems, since they will be too inefficient in time or space, but it is useful to go through the complete sequence so we can see how the practical solutions were developed.

17.3.6 Disk Block Pointers in the File Descriptor

We could keep all the disk block pointers in the file descriptor. The problem with this is that the file descriptor would be of variable size. This is not a feasible solution because it is too much trouble to keep track of variable-sized file descriptors.

Figure 17.8 shows file pointers kept in the file descriptor.

17.3.7 Disk Block Pointers Contiguously on Disk

We could keep the disk block pointers in a "mini-file" that is kept contiguously on disk. This has all the problems of contiguous files, albeit to a lesser extent because these files are much smaller. But modern file systems hold larger and larger files, and the problems will quickly recur if we choose this solution.

Figure 17.9 shows file pointers kept in a contiguous area of the disk.

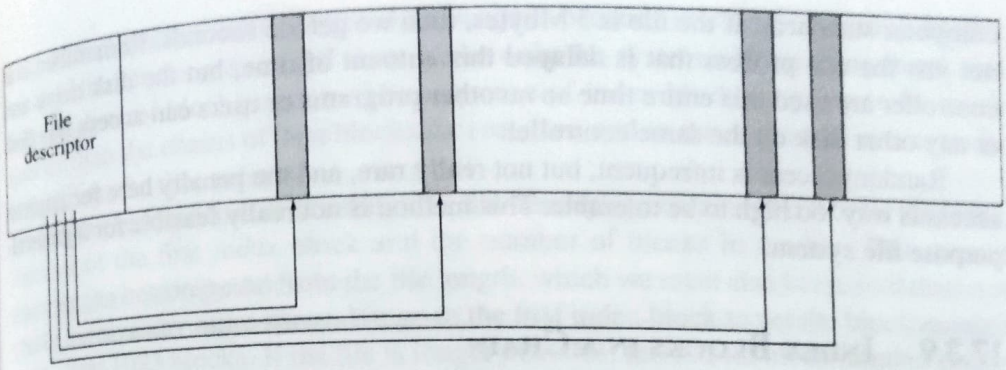


Figure 17.8 Block pointers in the file descriptor

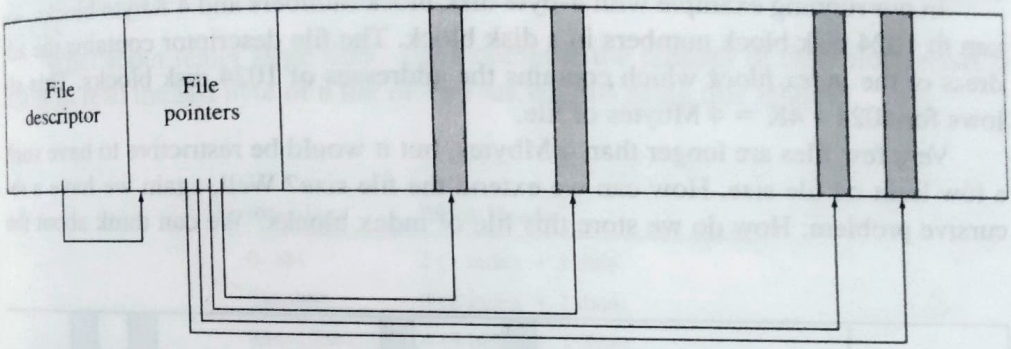


Figure 17.9 Block pointers in contiguous disk blocks

17.3.8 Disk Block Pointers in the Disk Blocks

A simple solution is to keep the pointer to the first logical block in the file descriptor, and, for each logical block in the file, the pointer to the next logical block in the current block. (See Figure 17.10.)

This method is simple, but it has several problems. The first problem is that you are taking four bytes away from each disk block. In memory, buffers will be the full disk block size, but four bytes will not be used. Users will have to remember that all the disk blocks are a word short. This is inconvenient, but not really that serious.

The second problem is much more serious. With this organization you must read the file sequentially, starting from the beginning, each time you use it. If you want to append to the end of the file (a common thing to do), you have to traverse the entire file to find the end. Random access to a file organized this way would be extremely expensive in terms of the number of disk block reads required.

Let's make some assumptions to get an idea of how inefficient it would be. Suppose it takes 10 milliseconds to read a disk block (this is very fast, even for the most modern disks). Suppose disk blocks are 4 Kbytes and we want to append to the end of a 300 Kbyte file. This will require reading 75 disk blocks (the entire file) and will take three quarters of a second, not that slow by human standards but very slow by

computer standards. If the file is 3 Mbytes, then we get 7.5 seconds. Remember, it is not just the one process that is delayed this amount of time, but the disk drive and controller are used this entire time so no other programs or users can access this disk or any other disk on the same controller.

Random access is infrequent, but not really rare, and the penalty here for random access is way too high to be tolerable. This method is not really feasible for a general-purpose file system.

17.3.9 Index Blocks in a Chain

We can save some time by collecting these block pointers together in a single disk block. Figure 17.11 shows how this would work.

In our running example with 4-byte disk block numbers and 4 Kbyte blocks, we can fit 1024 disk block numbers in a disk block. The file descriptor contains the address of the *index block* which contains the addresses of 1024 disk blocks. This allows for 1024 * 4K = 4 Mbytes of file.

index blocks

Very few files are longer than 4 Mbytes, but it would be restrictive to have such a low limit on file size. How can we extend the file size? Well, again we have a recursive problem: How do we store this file of index blocks? We can think about the

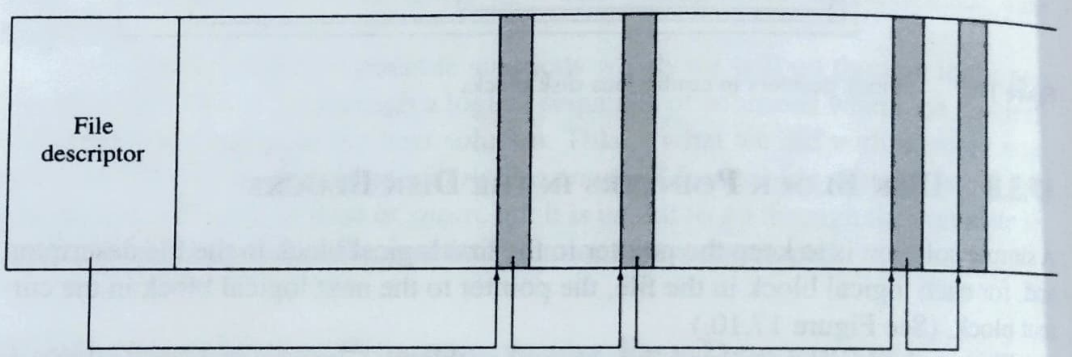


Figure 17.10 Block pointers in the blocks

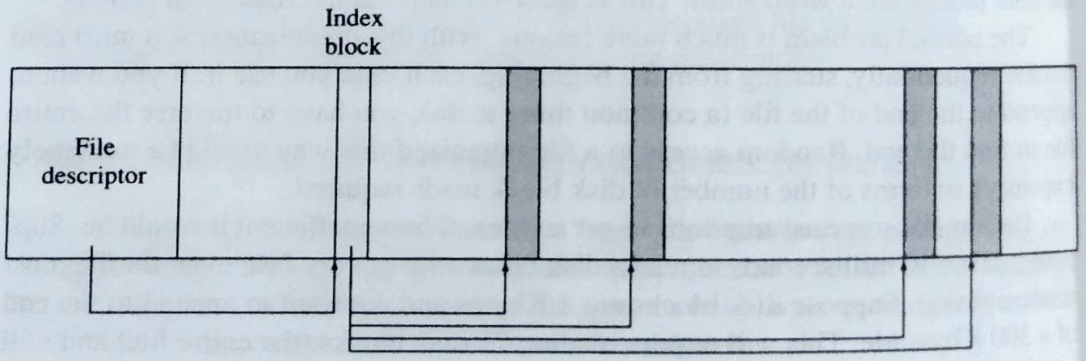


Figure 17.11 Block pointers in index blocks

linked blocks scheme in Section 17.3.8 again because this time it doesn't matter much if the blocks are a word short, since only the system will be using them, and only for lists of block pointers. The sequence of index blocks is going to be much shorter than the chains of data blocks for real files, and so the sequential access problem will not be that serious.

So the new scheme is like this: The file descriptor contains two words, the block number of the first index block and the number of blocks in the file. This second number can be computed from the file length, which we must also keep, so it does not really involve any more space. We go to the first index block to get the block number of the first 1023 blocks. If the file is longer than this, the 1024th block number points to a second index block. This gives us another 4 Mbytes (almost, because we lose one word from each index block for the next block pointer). If this isn't enough, we go to a third disk block, and so on. Figure 17.12 shows this idea with a chain of two index blocks.

What about really large files? Let's compute the number of disk block reads it takes to read the last byte of a file of various lengths using this scheme.

File Size	Block Reads
0–4M	2 (1 index + 1 data)
4M–8M	3 (2 index + 1 data)
8M–12M	4 (3 index + 1 data)
100M	26 (25 index + 1 data)
500M	126 (125 index + 1 data)
1G	251 (250 index + 1 data)
10G	2,501 (2,500 index + 1 data)
100G	25,001 (25,000 index + 1 data)

The growth is linear in the size of the file. One Gbyte is a really huge file by anyone's standards, and it takes 251 disk block reads, which is not too bad.

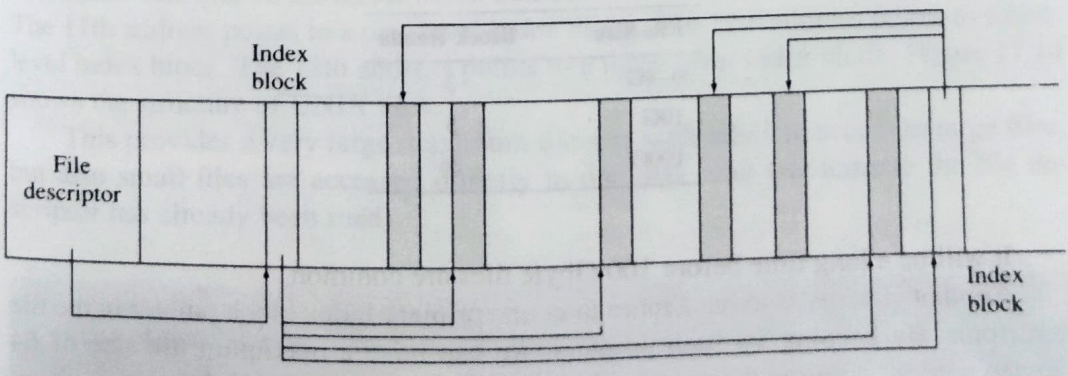


Figure 17.12 A chain of index blocks

There are ways we could improve this scheme with a little extra effort and storage. If appending to the end of the file was a common operation, we could keep another pointer to the last block in the chain. This would make appending very fast, no matter how large the file was. As another optimization, we could keep the index blocks on a doubly linked list and keep a pointer to the last index block we used. If file accesses show locality, then we can start from there and find new blocks without traversing the chain over from the beginning.

Linked index blocks is a practical method, but it is rarely used. The reason is that there are other schemes which are just about as simple and work better for very large files.

Index blocks collect the next-block pointers in one disk block.

17.3.10 Two Levels of Index Blocks

ble index blocks

Chained index blocks were a solution to the recursive problem: How do we store the "file" of index blocks on disk? To solve the problem, we went back to a previous solution, that is, chained block files. We could also solve the recursive problem by reusing the index idea again, that is, by using two levels of index.

The file descriptor contains a pointer to the primary index block. The primary index block contains the addresses of (up to) 1024 secondary index blocks. Each secondary index block contains the addresses of 1024 data blocks. (See Figure 17.13.)

Let's compute the maximum file size: 1024 primary index blocks * 1024 secondary index blocks * 4096 bytes in a data block = 4 Gbytes. This is a very large file and is an acceptable maximum size. In fact, if we use 32-bit unsigned words as file byte pointers, we can only have files 4 Gbytes long, so this is enough until we start using 64-bit integers for file pointers.

But even if we wanted a larger file size, we can chain the primary index blocks to get files of unlimited size. Let's look again at the number of disk block reads it takes to read the last byte of a file of various length using this new double-indexed scheme.

File Size	Block Reads
0–4G	3
10G	5
100G	26

It will be a long time before 100 Gbyte files are common.

Another strategy is to have more than one primary index block pointer in the file descriptor. By keeping 16 such pointers, we can have a maximum file size of 64 Gbytes without going to the extra complexity of chained index blocks.

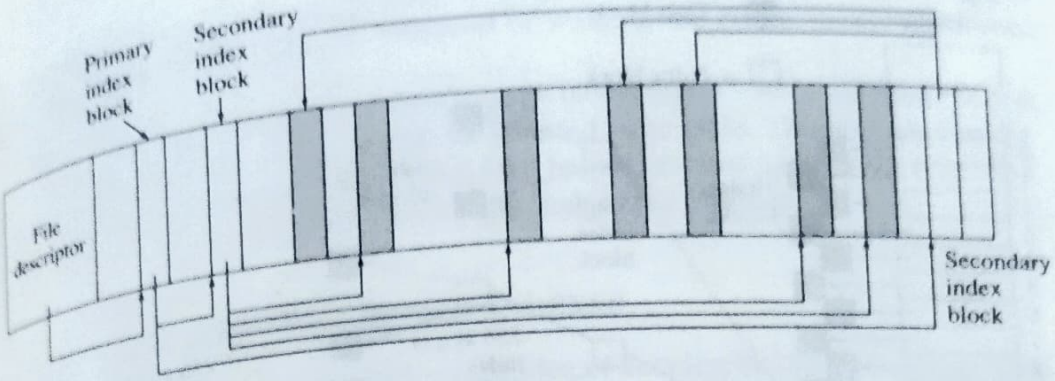


Figure 17.13 Two-level index

17.3.11 Large and Small Files

triple index block

Well, along the lines of "if some is good, more is better," we can go to three levels of index blocks. This gives us a maximum file size of 4 Tbytes, very large indeed. But more levels of index mean that the minimum time to access any data at all is longer. With triple index blocks, it will take four disk reads just to read the first byte of the file.

The problem here is that we are serving two masters. We want to be able to have very large files and access them efficiently, and we want to access small files quickly. All studies of file sizes in typical computer installations (see Section 17.6) show that small files predominate. In fact, the most common file size is one disk block. So, if we do not handle small files quickly, we will be giving up a lot of efficiency.

17.3.12 Hybrid Solutions

In this case, we can have the best of both worlds by combining these methods. Let us take the UNIX file organization as an example. The file descriptor contains 13 block addresses. The first 10 are direct block addresses of the first 10 data blocks in the file. The 11th address points to a one-level index block. The 12th address points to a two-level index block. The 13th address points to a three-level index block. Figure 17.14 shows the structure of UNIX files.

This provides a very large maximum file size with efficient access to large files, but also small files are accessed directly in one disk read (we assume the file descriptor has already been read).

Hybrid solutions are the best when the range of file sizes is very large.

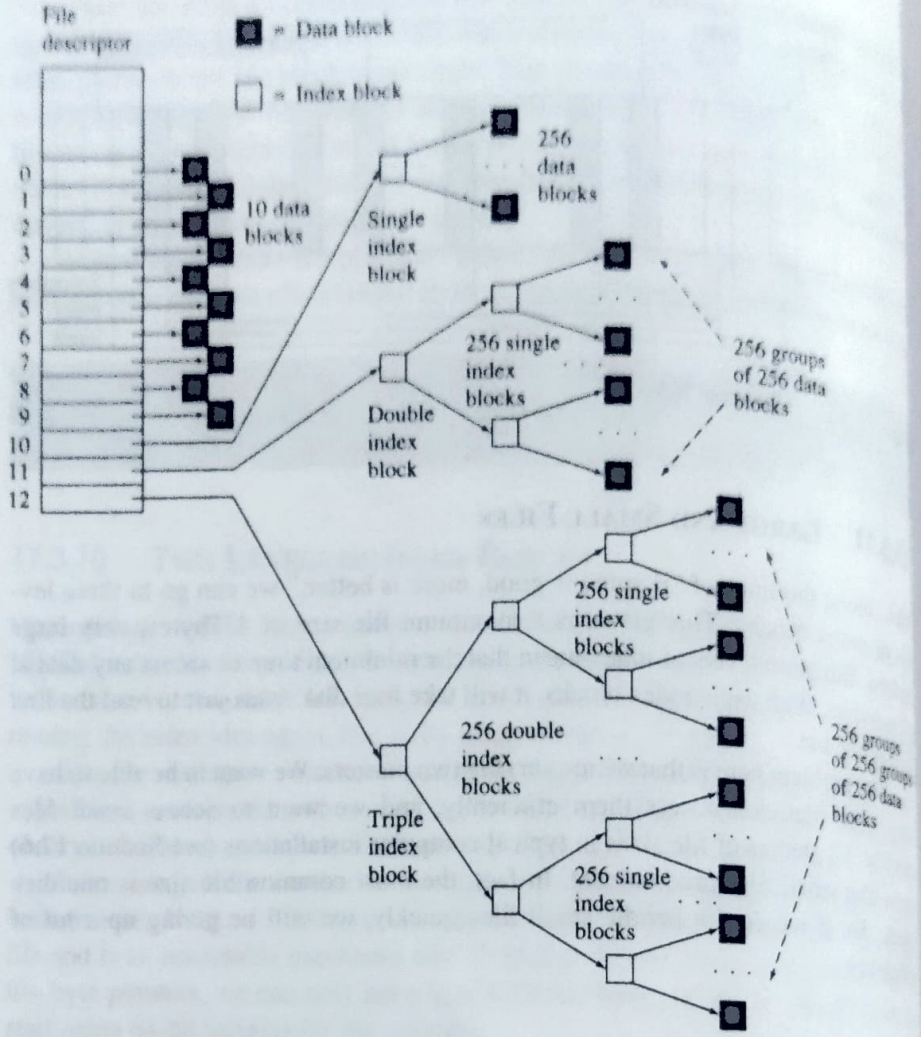


Figure 17.14 The UNIX file pointer structure

17.3.13 Analogy with Page Tables

You may have already noticed the analogy of these solutions with page tables in virtual memory systems. The reason for this is that disk block mapping and page mapping are solving the same problem, that is, keeping a list of addresses to the parts of a large amount of storage that is kept in small pieces.

In memory management, we considered and rejected contiguous storage just as we did for disk files. The drawbacks of contiguous storage allocation are just too great. The fact of fixed-size disk blocks leads us to paging-like solutions, and we tried one- two-, and three-level indexes, just as we tried one- two-, and three-level page tables. We have not tried variable-sized pieces, as we did in memory management, but we will explore that idea in a later section.

We did not try chaining solutions in page tables since random access is too common in page tables, and the overhead of walking the chains would have been way too high.

There is one solution that we used in page tables that we have not tried for disk block pointers, and that is the idea of an inverted page table. The next section discusses that idea. This idea was developed long before inverted page tables were tried, but they both represent the same design approach to the problem.

17.3.14 Inverted Disk Block Indexes

inverted disk block indexes

Up to now, we have been committed to the idea of keeping the file pointers for each file separate. Another plan would be to keep all the file pointers together in one big file. How big does this file have to be? Let's think of it this way. Suppose we think back to the linked file block idea. Each disk block contained the pointer to the next disk block in the file. There is exactly one block pointer for every disk block. Suppose the disk contains 10,000 blocks. Then we will have a file of 10,000 block numbers. This file will never change in size since it is based on the size of the disk itself, not the size of any particular file on the disk. If we have 4 Kbyte blocks and require 4 bytes for each block pointer, we can fit 1024 block pointers in one block, and so these 10,000 block numbers will fit in 10 disk blocks.

So here is the plan. The first 10 disk blocks contain all the next-block pointers for all the blocks on the disk. We still have to chain through the disk blocks sequentially to get to the end of the file, but all the disk blocks are together so this will take no more than 10 disk block reads. In fact, we can keep these 10 disk blocks in memory, and all file pointer chaining will be done by following pointers in memory. Compared to disks, main memory is very fast, so this will take almost no time at all.

Figure 17.15 shows the structure of an inverted disk block index. The inverted index is on disk and consists of a large array containing one disk block pointer for each (allocatable) disk block. We chain disk blocks together in a list by chaining their corresponding disk block pointers in the inverted index. In Figure 17.15, we show the chain for one file. The file contains five disk blocks. It starts at the block labeled *bof* (beginning of file) and ends at the block labeled *eof* (end of file).

This approach is very good for small disks but it breaks down for larger disks. Let's do some calculations. Suppose we have 4K disk blocks and 4-byte block numbers (as usual). Here is a table of the amount of memory and the number of disk blocks required for various sizes of disk:

Disk Size	Memory	Disk Blocks
40 Mbyte	10K	3
240 Mbyte	60K	15
4 Gbyte	1M	256

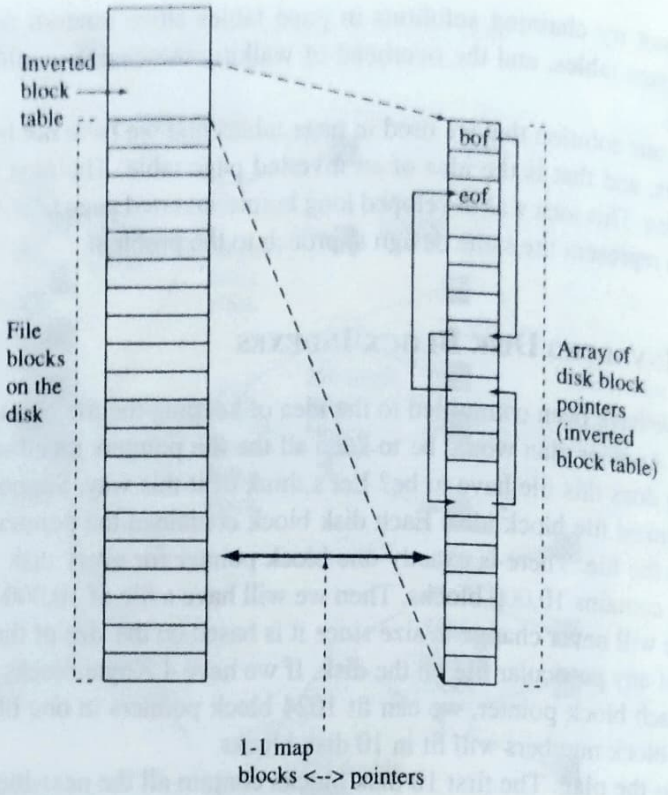


Figure 17.15 An inverted disk block index

The disk space overhead is always 0.1 percent, which is acceptable and in fact quite good. But the memory use goes up too quickly. For this method to work well, the inverted block table must be in memory (or mostly in memory). The reason for this is that our old friend locality is not present in this case. It is not usually the case that logically consecutive disk blocks are in physically consecutive disk blocks, and so we will jump all around the inverted index when following the blocks of a file.

Now 10K of memory for a 40M disk is acceptable, and even 60K for a 240 Mbyte disk given the large memories available today. But suppose we have three 4-Gbyte disks on a file server. This means that 3 Mbytes of memory is taken up for the inverted indexes. This is too much memory to allocate to this use because main memory is the most critical resource in modern computer systems. There are lots of better uses for that memory, since the other disk organizations we discussed work just as well (or better) and do not demand so much memory space.

What is the problem here? Why is locality not helping us? The reason is that this method ignores the critical locality that we need to take advantage of. This method does not discriminate between files that are currently being used and those that are not. The index block methods will keep block indexes in memory also, *but only for files that are currently open*. This makes all the difference because a computer system typically has lots and lots of files, but only a tiny fraction of them are open at any one time. It is this locality that the other disk organization methods are able to take

advantage of and give fast disk access with a minimal use of main memory for file pointers.

The inverted disk block index method is used in the MS/DOS file system. MS/DOS system started out as a floppy-disk-oriented system. This method works very well for floppy disks, since they are small. A 360K floppy with 512-byte sectors will have 720 blocks. Since two bytes is enough per block, we need only a little over 1K to hold the inverted block index. Also, MS/DOS combined disk blocks to get an effective block size of 4K and reduced the size of the inverted block index by a factor of 8.

17.3.15 Using Larger Pieces

Up to now, we have assumed that the pieces we break the file up into are the same size as the disk blocks defined by the disk itself. There are three ways we can move beyond this:

- Use larger blocks.
- Allocate blocks several at a time.
- Use variable-sized pieces.

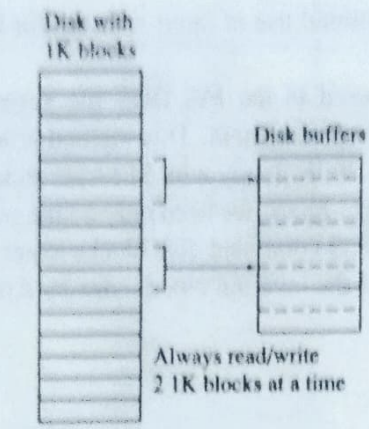
We can ignore the block size of the disk and use some multiple of it. For example, suppose that the disk had 1K blocks but we wanted to use 4K blocks. We could always read and write blocks four at a time, and act as if the block size of the disk was 4K. Many operating systems will do this when using disks with small block sizes. This technique is known as *clustering*.

clustering

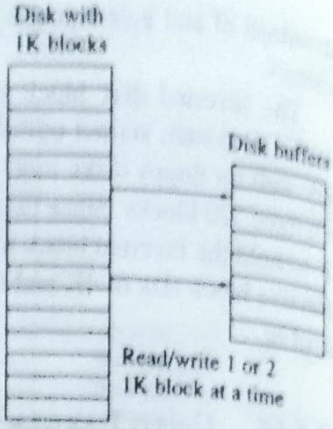
It is more efficient to read disks in larger units. We discussed this in Section 14.4.1. The layout and access characteristics of disks imply that it only takes a little longer to read four disk blocks than it takes to read one disk block. For example, it might take 20 milliseconds, on the average, to read one disk block, and only 22 milliseconds to read four disk blocks. The actual transfer time is only a small part of the total disk access time.

A variation of this idea is still to read and write single disk blocks, but to allocate space on the disk in multiples of two or more blocks. For example, we might use 1K disk blocks, but always allocate disk space in units of four blocks at a time. This ensures that each four-block chunk will be allocated to consecutive disk blocks and will be fast to read. Note that disk blocks still take the same amount of space in main memory. We are simply ensuring that consecutive disk blocks will be read quickly from the disk.

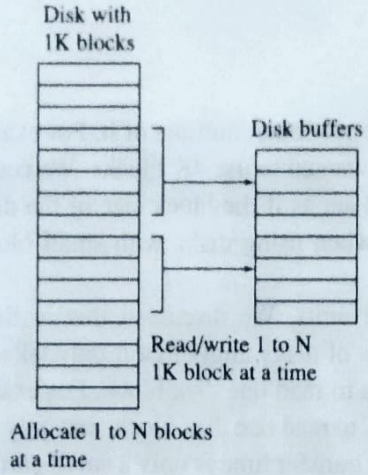
Figure 17.16 shows the three ways to use larger pieces of the disk than a single disk block. In Figure 17.16(a), we combine pairs of 1K physical disk blocks into 2K logical disk blocks. All reading, writing, allocation on disk, and disk buffers are 2K long. In Figure 17.16(b), we keep the logical block size at 1K, the same as the physical block size, but we allocate physical blocks two at a time. Disk buffers are still 1K long, and reads and writes can be 1K or 2K long. This allows us to read files two blocks at a time for higher-speed access.



(a) Logical disk block of 2K



(b) Logical disk block of 1K (allocate in 2K pairs)



(c) Allocate in variable sized extents

Figure 17.16 Three ways to use larger pieces than one block

17.3.16 Variable-Sized Pieces

Another way to use larger pieces is to allow variable-sized pieces on disk. In a variable-sized piece scheme, we would not just keep a list of block addresses, but a list of pairs: block address and number of blocks. We would use as many contiguous blocks as we could for each piece. Variable-sized pieces are also called *disk extents*.

In Figure 17.16(c), we show extents where we allocate in units of 1 to N disk blocks (as many as we need and can find together). Disk buffers are still 1K, the physical block size. But we can read and write 1 to N blocks at a time.

The advantage of this method is that we could keep files physically sequential for longer sequences, and enjoy the faster disk access this implies. Also, we would need fewer descriptors if each one described more of the file.

Unfortunately, there is a problem with this method (isn't there always something?). The problem is that, in order to access the file randomly, you have to go through all the pieces and count up the blocks. You must do this because each extent may have a different size. Also, space management is harder because we need to detect and allocate consecutive sectors whenever possible. As a consequence, the method of variable-sized extents is rarely used.

The following code demonstrates this problem by showing the two versions of the function that converts a logical block number to a physical block number, one using single blocks and one using variable-sized extents. The array that does this conversion (for the case where we allocate single blocks at a time) is scattered around the disk in index blocks and whatnot. To make the algorithm simpler, we are assuming that this array exists as a single array in memory. This makes it easier to compare the single-block method with the extent method. If we used extents, then the extent array would also be scattered around the disk. Again, we assume it is in a single array available to the translation function.

First, we present the algorithm for fixed-sized extents.

RANDOM ACCESS WITH FIXED-SIZED EXTENTS

```
// Assume some maximum file size
#define MaxFileBlocks 1000
// This is the array of logical to physical blocks
DiskBlockPointer LogicalToPhysical[MaxFileBlocks]
// This is the procedure that maps a logical block number into a
// physical block number.
DiskBlockPointer LogicalBlockToPhysicalBlock (int logicalBlock) {
   // Just look the physical block number up in the table.
   return LogicalToPhysical[logicalBlock];
}
```

The procedure consists of just a return statement that does an array access.

And now we present the algorithm for variable-sized extents.

RANDOM ACCESS WITH VARIABLE-SIZED EXTENTS

```
// Assume some maximum file size
#define MaxFileBlocks 1000
// This is the type for extent structures
struct ExtentStruct {
   DiskBlockPointer baseOfExtent;
   int lengthOfExtent;
```

```
};
// This is the table of extents.
ExtentStruct Extents[MaxFileBlocks];
// This is the procedure that maps a logical block number into a
// physical block number.
DiskBlockPointer LogicalBlockToPhysicalBlock( int logicalBlock ) {
   // Compute the logical block number of the first block of each extent.
   int lbOfNextBlock = 0;
   // Loop through the extents.
   int extent = 0;
   while( 1 ) {
      // Figure out the logical block number of the first block of
      // the NEXT extent (not this one).
      int newlb = lbOfNextBlock + Extents[extent].lengthOfExtent;
      // If the next extent is too far then the logical block we
      // are looking for is in this extent, so exit to loop.
      if( newlb > logicalBlock )
         break;
      // Move to the next extent.
      lbOfNextBlock = newlb;
      ++extent;
   }
   // The physical block is an offset from the first physical block
   // of the extent.
   return Extents[extent].baseOfExtent + (logicalBlock - lbOfNextBlock);
}
```

Notice that the first case is very easy. It is a table lookup with no loop. The second case is harder because we have to loop through the extents until we find the one our block is in. As we loop through, we keep a running block count.

Note that looking through all the extents each time not only takes processor time, but it means that the entire extent table needs to be in memory all the time.

17.3.17 Disk Compaction

Several of these variant techniques we have been considering have one purpose: to get the blocks in a file in physically consecutive sectors on the disk. Our basic method allocates disk blocks one at a time anywhere on the disk. When it comes time to allocate another block, it is unlikely that you will allocate the physically consecutive one even if it was free (which it probably isn't). This means that files are scattered around the disk, and it takes longer to read a file sequentially, which is the usual mode of file access.

But although the method does not require logically consecutive blocks to be physically consecutive, it does not prevent this either, and this provides us with a way to improve access speed. Suppose one night we rearrange the disk so that all the files are in physically consecutive blocks. This is easily done by reading blocks, buffering them in memory while the disk is rearranged, and rewriting files out in physically consecutive blocks. This will take a while, but once we have done it, these files will stay physically consecutive (unless they are extended). If you do this every night, most of the files that stick around will be physically consecutive and will be quick to read.

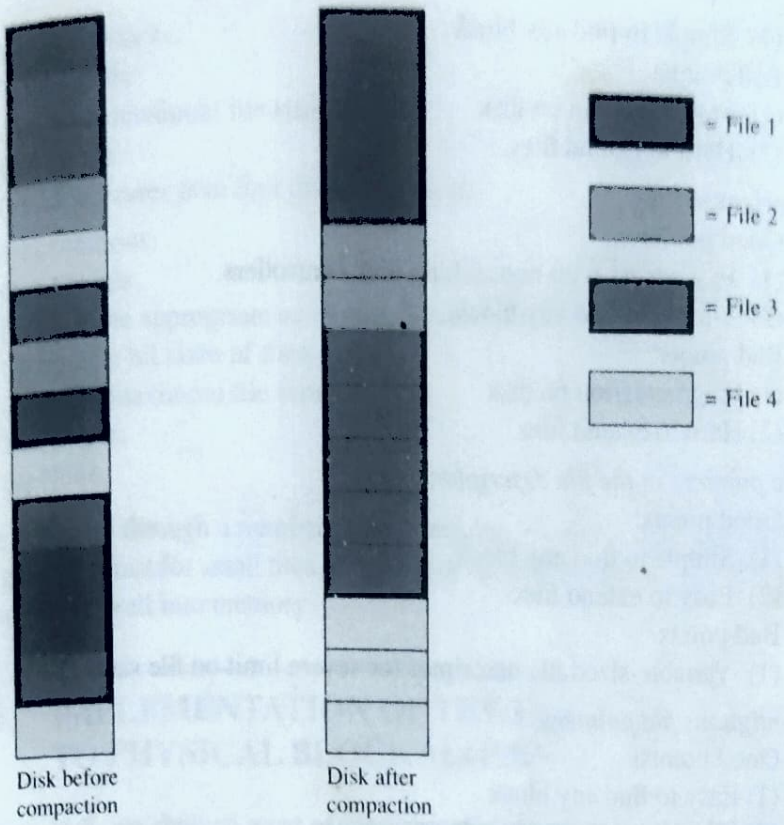


Figure 17.17 Disk compaction (for four files)

This process is called *disk compaction* and is commonly done in file systems. It only helps with files that stay around for a while, but that includes a lot of the file traffic (include files, system binaries, libraries, etc.). So again we are getting the best of both worlds: a file implementation that allows disks blocks to be anywhere on the disk and where allocation is easy, and a system where the most commonly used files are kept in consecutive blocks.

disk compaction

Figure 17.17 shows how disk compaction works. The four files shown have their blocks scattered around the disk. After compaction, the blocks of a file are always contiguous and so can be read very rapidly.

Disk compaction can get the benefits of contiguous files within most file organizations.

17.4 REVIEW OF FILE STORAGE METHODS

Here is the sequence of attempts we made:

1. *Contiguous files*—Keep all data blocks contiguously on disk.
   a. Good points:
      (1) Fast access.