expensive and is the main demerit of a bit map over a free blocks list. However, a bit map has the advantage of being very simple to implement. The first look may suggest that the block list method will occupy much more memory than the bit map method. But this is deceptive. The reason is that, normally, the free blocks list itself is kept in free blocks and, therefore, does not require any extra space.

Non-contiguous Allocation

In non-contiguous allocations, the maximum size of a file does not have to be predicted at the beginning. The file can grow incrementally with time as per the needs. This gives it a lot of flexibility at reduced wastage of disk space. Another advantage is that the Operating System automatically allocates additional blocks if the file gets full during the execution of a program, without aborting the program and without asking for the operator's intervention. But then, the Operating System routines to handle all this become more complex. There are two main methods of implementing non-contiguous allocations, viz. **chained allocation** as in MS-DOS or OS/2 and **indexed allocation** as in UNIX, AOS/VS or VAX/VMS. We will consider these one by one.

(i) Chained Allocation Chained allocation believes in allocating non-contiguous blocks to a file. But then, the Operating System must have a method of traversing to the next block or the cluster of blocks allocated to that file, so that all the blocks in a file are accessible, and therefore, the whole file can be read in sequence. A pointer is a field which gives the address of the next block(s) in the same file. This address could comprise block number instead of the physical address as we have seen. One of the early ideas was to reserve two bytes in a block of 512 bytes to give this address. This scheme is not very popular today but we will start our description with it.

In one of the schemes of chained allocation, the following happens (Windows 2000 follows a slightly different version of this as we shall see a little later).

(a) The file directory entry gives the first block number allocated to this file. For instance, Fig. 4.36 shows that the first block for 'FILE A' is 4.

(b) A fixed number of bytes (normally 2) in each block are used to store the block number of the next block allocated to the same file. We will call it a pointer. This means that in a 512 byte block, only 510 bytes can be used to store the data, as 2 bytes are used for a pointer.

(c) With 2 bytes, i.e. 16 bits to denote the block number, the maximum blocks on the disk and therefore in a file can be 2^{16} or 65536 or 64K. Therefore, the maximum file size would be 32 MB (1 block = 512 bytes = 0.5 kB). However, out of this, the actual data is obviously slightly less. It will actually be $510 \times 32/512$MB.

(d) Some special characters with predefined ASCII codes used as a pointer in the last block for a file, indicate the end of the chain. This is shown as '*' in Fig. 4.36.

For instance, if blocks 4, 14, 6, 24 and 20 are allocated to a file, they will be chained as shown in Fig. 4.36.

A better scheme, as followed in Windows 2000 or OS/2, is to keep these pointers externally in a **File Allocation Table (FAT)** as shown in Fig. 4.37. In this scheme, the block can have full 512 or 1024 bytes of actual data, depending upon the block length (In MS-DOS, the block length is 1024 bytes). You still have the overhead of the FAT which is again 2 bytes per pointer and therefore, per block. The only difference is that in FAT, the pointers are kept externally.

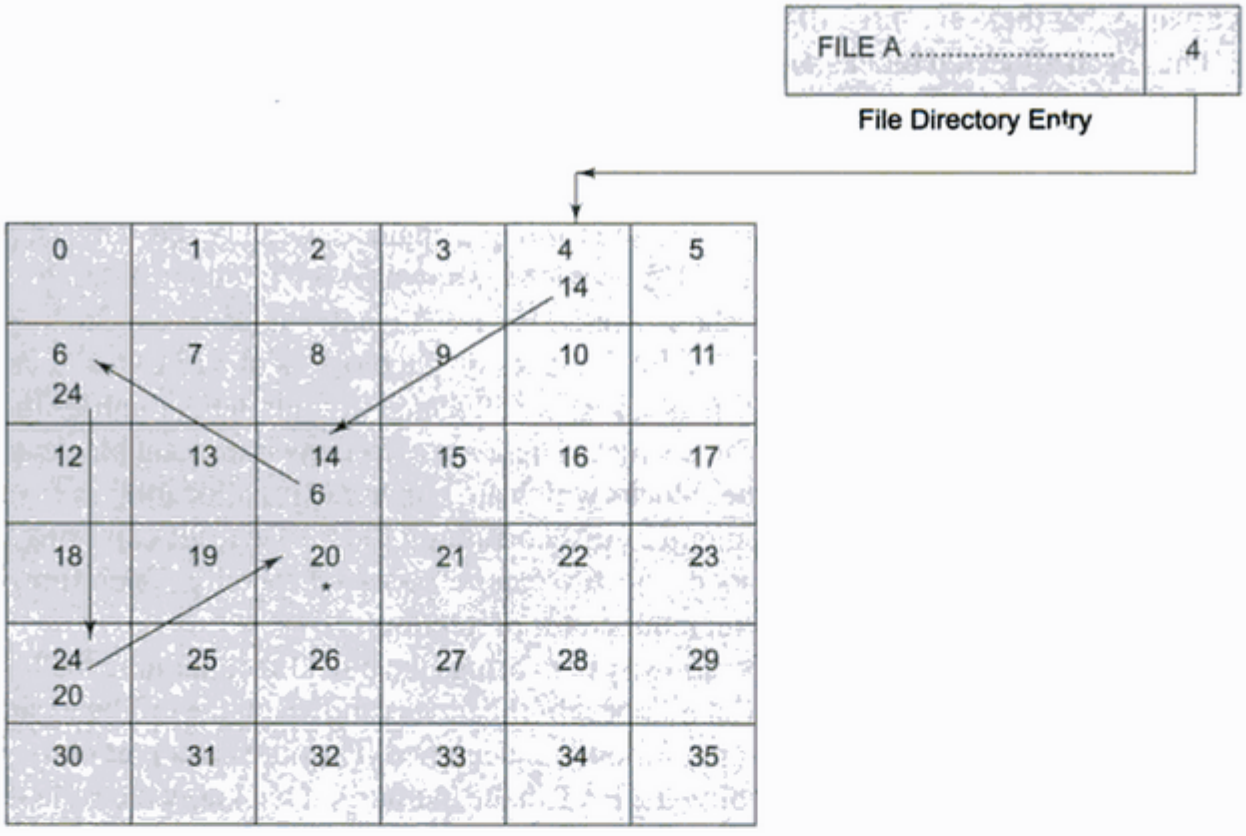

Fig. 4.36 Chained allocation

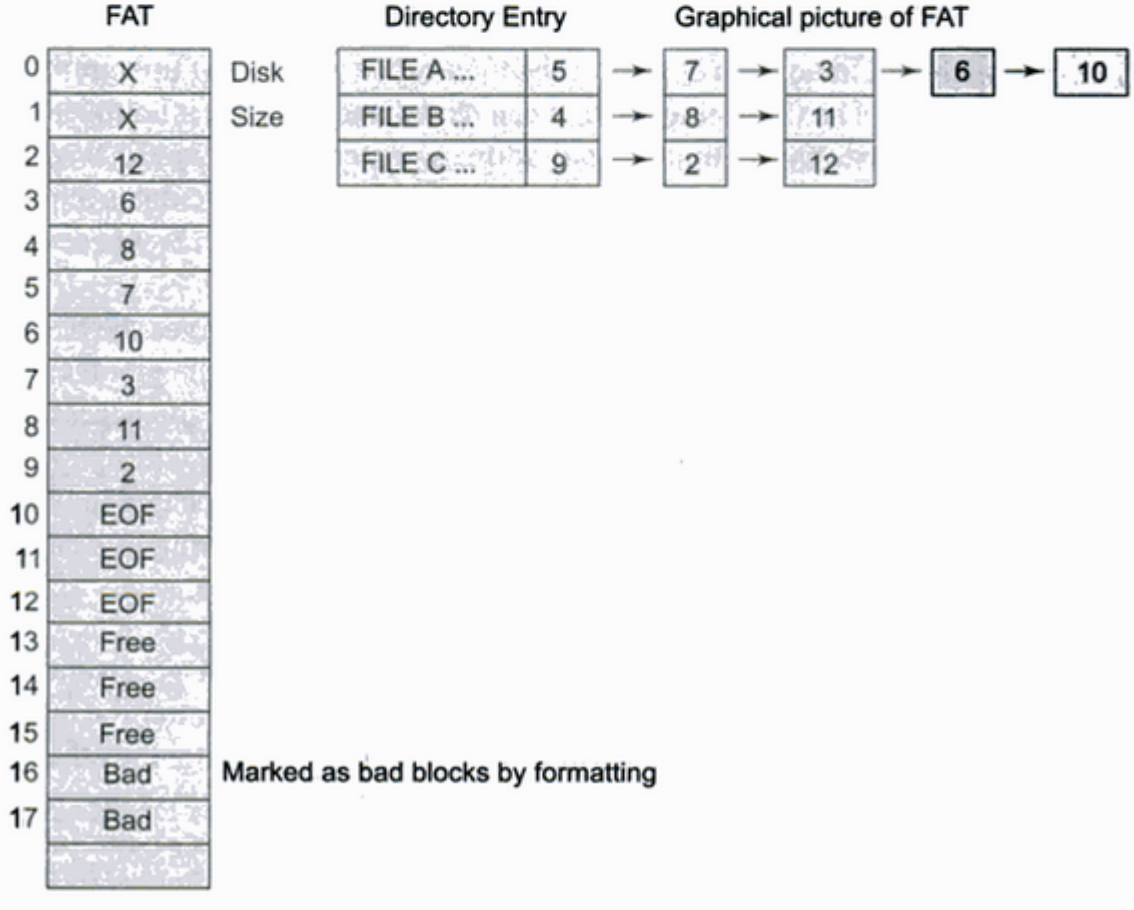

Fig. 4.37 MS-DOS, Windows 2000 or OS/2 chained allocation

Let us assume that there are three files in our system. FILE A has been allocated blocks 5, 7, 3, 6 and 10. FILE B has been allocated blocks 4, 8 and 11, and FILE C has been allocated blocks 9, 2 and 12. The file directory entries for these three files have been shown in the figure which mention the first block number in these respective files, viz. 5, 4 and 9. Against the directory entry on the right side is the list of blocks allocated to different files which is shown only for our understanding, because the actual blocks allocated to that file are maintained in the FAT in the form of a chain as we shall see. This is, in a sense, a conceptual and a graphical representation of FAT which is shown on the left hand side.

The study of FAT will reveal how the chain works. Each FAT entry is of fixed size, say 2 bytes. Therefore, a block of 1024 bytes can have 512 FAT entries or if a block is of 512 bytes, it can contain 256 FAT entries. Therefore, normally specific block(s) are allocated to contain FAT entries themselves, depending upon the total number of blocks on the disk. These are extremely important blocks on the disk which contain the information about all other blocks which are either bad (unallocable), free (allocable) or already allocated to different files. A computer virus corrupting the FAT entries can bring the whole system to a standstill, because, if this happens, no file can be accessed reliably. Therefore, normally more than one identical copies of FAT are maintained for protection.

Before any file is read or written, the Operating System brings the block(s) containing the FAT entries in the memory for faster operation. The serial numbers of the blocks shown on the left hand side of FAT in Fig. 4.37 and outside the box are shown only for our convenience. They are not a part of the FAT, and therefore, are not actually stored. After reading the FAT in the memory, if we know the Byte Address of the beginning (i.e. the zeroth entry) of the FAT (say A), the address of any entry can be found as $A + 2n$, where n = entry number. This is because, each entry takes 2 bytes. Hence, there is no reason for storing them.

Suppose that we want to read FILE C sequentially. The procedure will be fairly straightforward. The file directory entry which will have been copied to the memory at the time of opening the file gives the first block number which is 9. The Operating System can read that data block after the necessary address conversions/translations into its three dimensional address. After this, it would read the next block allocated to this file. To do this, it treats this 9 as a random key and then it calculates the address of entry number 9 of FAT in the main memory, by the formula $A + (2 \times 9) = A + 18$, where A is the starting address of the FAT.

Now the Operating System accesses entry number 9 in the FAT which gives the next block number allocated to this file as 2. We can verify this in the FAT (9th entry) and also the graphical representation on the right side. The Operating System now can read this data block after the necessary translations. After this, the Operating System accesses entry number 2 of FAT at memory address of $A + (2 \times 2) = A + 4$. This entry gives the next block number as 12 as the third data block allocated in this file (i.e. FILE C). The Operating System can now read block number 12. Again the Operating System treats 12 as the random key and accesses the 12th entry of the FAT at memory address of $A + (2 \times 12) = A + 24$. This entry says "EOF", i.e. End of File.

This indicates that there are no more blocks for this file. This is how, if you want to read FILE C sequentially, the Operating System can read block numbers 9, 2 and 12 one after the other. The point is that the AP normally reads logical data records (e.g. Customer record) one after the other. Therefore, the conversion of a logical data record to blocks still needs to be done. This conversion is done to logical blocks (0, 1, 2 in this case) first and then to physical blocks (9, 2, 12 in this case). For instance, when the AP reads a very first data record (RRN = 0) of 700 bytes in a system where block consists of 512 bytes,

the file system will have to read all 512 bytes from LBN = 0 (i.e. PBN = 9) and the first 188 bytes from LBN = 1 (i.e. PBN = 2) We will take another example to illustrate this.

How will our previous example of a sequential processing with "While (!eof) – fread" in C or "READ CUST-REC... AT END" in COBOL work in this case? It is easy to imagine. Most of the processing is absolutely similar to what we have described earlier, until we arrive at the logical block numbers (LBN) to be read. In our example in Section 4.2.5, we had to read the following to get one CUSTOMER record of 700 bytes starting at RBN = 1400 (refer to point (iv) in Section 4.2.5 and Fig. 4.20).

Logical block no. = 2, Last – 136 bytes Logical block no. = 3, All – 512 bytes Logical block no. = 4, First – 52 bytes Total 700 bytes

In our example, if FILE A, shown in Fig. 4.37 is the CUSTOMER file, the logical blocks 2, 3 and 4 are the 3rd, 4th and the 5th blocks from the beginning (because logical block 0 is the first block in the file). These are physical block numbers 3, 6 and 10. Therefore, the Operating System will have to read blocks 3, 6 and 10 in the controller's memory, form a logical record of 700 bytes as shown above and then transfer it to the I/O area of the AP. The reading of these blocks 3, 6 and 10 one after the other is obviously facilitated due to the chains maintained in the FAT. You can now easily imagine how the Operating System can satisfy the AP's requests to read logical customer records sequentially one after the other by using an internal cursor, and traversing through these chains in the FAT.

With chained allocations, online processing however tends to be comparatively a little slower. A Data Management System (DMS) used for an online system will have to use different methods for a faster response. An index shown in Fig. 4.22 is one of the common methods used in most of the Relational Database Management Systems (RDBMS). Imagine again that we have an index as shown in the figure. An Application Program (AP) for "Inquiring about customer information" is written. A user asks for the details of a customer with customer number = "C009". How will the query be answered? Let us follow the exact steps.

(a) The AP will prompt for the customer number for which details are required.
(b) The user will key in "C009" as the response.
(c) This will be stored in the I/O area for the terminal of the AP.
(d) The AP will supply this key "C009" to the DMS to access the corresponding record (e.g. MS-Access under Windows 2000).
(e) DMS will refer to the index and by doing a table search, determine the RBN as 700 (refer to Fig. 4.22).
(f) DMS now will request the Operating System, through a system call, to read a record of 700 bytes from RBN = 700.
(g) The Operating System will know that it has to read 700 bytes starting from Relative Byte Number = 700 (after skipping the first 700 bytes, i.e. 0–699). This gives us the starting address = 700/512 = 1 + 188/512, i.e. the reading should start from byte number 188 of Logical Block Number (LBN) = 1. But only 511 – 187 = 324 bytes will be of relevance in that block. Therefore, we would need 700 – 324 = 376 bytes from the next block (i.e. LBN = 2)

(h) The Operating System will translate this as:

LBN	1: Last 324 bytes (188–511)
+ LBN	2: First 376 bytes (0–375)
	Total : 700 bytes

(i) In our example, if FILE A is the CUSTOMER file, as per the FAT, logical block 0 (given in the directory entry) = physical block number 5. Similarly, logical block 1 will be physical block number 7 and logical block 2 will be physical block number 3 (refer to Fig. 4.37). Therefore, the DD issues instructions to the controller to read physical blocks 7 and 3, pick up the required bytes as given in point (h) and formulate the logical record as desired by the AP.

(j) The Operating System transfers these 700 bytes to the I/O area of the AP (perhaps through the DMS buffers, as per the design).

(k) The AP then picks up the required details in the record read to be displayed on the screen.

This is the way the interaction between the Operating System such as Windows 2000 and any DMS such as MS-Access takes place.

An interesting point emerges. In this method, how does the Operating System find out which are the logical blocks 1 and 2? The Operating System has to do this by consulting the file directory entry, picking up the starting block number which is LBN 0. It then has to consult the FAT for the corresponding entry (in this case entry number 5) and proceed along the chain to access the next entry each time adding 1 to arrive at the LBN and checking whether this is the LBN that it wants to read. There is no way out. If logical block numbers 202 and 203 were to be accessed, the Operating System would have to go through a chain of 202 pointers in the FAT before it could access the LBN = 202 and 203, get their corresponding physical block numbers and then ask the controller to actually read the corresponding physical blocks.

If we had the pointers embedded in the blocks, leaving only 510 bytes for the data in each block, the chain traversal would be extremely slow because the next pointer would be available only after actually reading the previous block, thereby, requiring a lot of I/O operations which are essentially electromechanical in nature. If the FAT is entirely in the memory as in Windows 2000 or OS/2, the chain traversal is not very slow because, you do not have to actually read a data block to get the address of the next block in a file. However, as the chain sizes grow, this is not the best method to follow, especially for online processing as we have seen. This is the reason why indexes are used for disk space allocations in some other Operating Systems.

(ii) Indexed Allocations An index can be viewed as an externalized list of pointers. For instance, in the previous example for FILE A, if we make a list of pointers as shown below, it becomes an index. All we will need to do is to allocate blocks for maintaining the index entries themselves and the file directory entry should point towards this index.

5	7	3	6	10

The problem is how and where to maintain this index. There are many ways.

CP/M Implementation CP/M provides an easy solution. It reserves space in the directory entry itself for 16 blocks allocated to that file as shown in Fig. 4.38.