DISCS – THE SOFTWARE VIEW

Disc management resembles memory management in that there are distinct views from the standpoints of the consumers and the provider. This chapter presents the consumer's view; the provider's view is inspected in the next chapter.

WHAT ARE DISCS USED FOR ?

That depends on the system. Four sorts of material that might be found on a typical system disc are presented in the list below. All these are likely to be found in a conventional large system, batch or interactive. Smaller systems such as microcomputer systems might miss out the two middle items, though the more ambitious microcomputer systems might include them all nowadays. In each case, we've added some descriptive notes which might help to decide what sort of administrative structures are useful for storing the material.

Function	*Description*	*Properties*			
		Stability	*Frequency*	*Structure*	*Size*
STORE : ORDINARY FILES	Medium-term or long-term memory.	changes only slowly.	used now and then.	directory structures.	(may be very) large.
SPOOLING	Files in transit, typically to or from another device or system.	steady change.	fairly active.	set of queues.	medium to large size.
SWAPPING : VIRTUAL MEMORY	An extension of primary memory.	rapid change.	very active.	segments or pages.	medium sized.
SYSTEM CODE AND DATA	Must be there; used by everybody.	permanent.	active.	not much.	smallish.

All these ways of using the disc have different characteristics, and therefore are most appropriately represented by different data structures. We've suggested appropriate structures in the material below; take these as hints rather than as hard requirements. It's the software's job to implement these required structures using the rather limited structure built into the disc hardware. (Compare memory models.)

"ORDINARY" FILES.

Storage management for ordinary files on the disc is much like segmented memory management. Because of the disc structure, space is available in fixed-size areas called sectors, rather like pages, but there are good reasons why this might not be the best unit for allocating space. As we noticed in the *DEVICES* chapter, it can be helpful to allocate segment-like groups of contiguous sectors, so file systems may make provision for such variable-sized requests. In some systems, the desired size of the allocation unit as a number of sectors can be specified as a file attribute for each file.

As with memory management, the allocation software has to keep track not only of the allocated space but also of which parts of the disc are available for allocation. The amount of memory used to hold the necessary information is a significant problem. It is possible to find methods which don't need much memory; here are two examples :

- **Incremental allocation with compaction** is the simplest way : keep all used sectors at one end of the disc. All you need to know is the disc address of the first sector of each file and of the first vacant sector, so the method uses almost no

memory space. It works fairly well for *very* simple systems, but doesn't for discs shared by many processes. The overhead is considerable, as any time a file is deleted, all following files must be shuffled backwards, and the directory modified accordingly. (Alternatively, you can remove deleted files from view, but defer reorganisation until the disc is full, then remove all the deleted files at once; that gives a bonus in the possibility of retrieving deleted files, if you're lucky, but it's still expensive.)

- **Link the unused areas**, as in segmented memory management. This method also requires very little memory, as all the geographical information is on the disc itself. The method works reasonably well if you only allocate one sector at a time, when any vacant sector will do, but if larger contiguous chunks are required it becomes expensive, as requests for large chunks may require many disc reads.

Despite their low demands for memory, these methods are not satisfactory for general use, so we have to give away some memory if we want an effective disc system. A sector map of *available space* must be provided – this time, not quite like segmented memory management, as in the interests of maintaining reasonable speed the map should preferably be held in memory rather than on the disc.

At its simplest, this *disc map* is an area of memory in which one bit is allocated to represent each disc sector. (Compare the page map used in managing paged memory.) The state of the bit denotes the availability of the sector. This is the cheapest way to keep track of disc allocations by sector. The information is compact (at 1 bit for perhaps a kilobyte of disc) and - if all you want is to know where to find vacant sectors - complete. In practice, as with paged memory, it is usually advantageous to keep more information about the sectors, and a few bytes per sector is a typical size. Information maintained varies from system to system; one useful item is a flag marking a bad disc sector, so that limited physical damage to the disc does not necessarily render it useless. In MS-DOS and related systems, this is the File Allocation Table (FAT), and also holds the links between sectors allocated to the same file which we saw in the previous chapter were necessary to permit flexible file management.

A more compact method has advantages with disc systems storing gigabytes or more of data, when the disc map can be larger than is convenient. Many systems therefore use a representation based on *clusters* of sectors – clumps of a few sectors each – as the unit of allocation; this stratagem reduces the demands on memory at the cost of some waste of disc.

The amount of disc space wasted depends on the sizes of the files – and particularly on the number of disc sectors occupied by a typical file. The university's computing service for students used to be supported (which not everyone agreed was the appropriate word) by a DEC-10 computer running the Tops-10 operating system, which uses clustering in its file system. For purely financial reasons, the system was chronically short of disc and memory space, so we were eager to find the most economical cluster size. A complete listing of the whole disc system directory was obtained around the middle of the year, when by more or less specious arguments it was expected that the distribution of file sizes would be reasonably representative of the long-term pattern. Then, using these statistics, the total disc space used for a range of cluster sizes was calculated. It turned out that there was a rather broad minimum, with the best result corresponding to three-sector clusters.

So that's what we used. But there is a catch. In the Tops-10 file system, every cluster has two additional sectors appended, one at each end, with information used by the system. Our three-sector clusters therefore used five sectors each, so the maximum possible

useful storage capacity of the disc was 60% of its actual capacity. Further, most of the files were about three sectors long; it follows that even when a cluster was completely full (because it belonged to a file of more than three sectors) it was quite likely that the next cluster in the file would be almost empty. The real occupancy was probably somewhere around 50%.

The reason was that almost all the files were small. The system was used almost exclusively by students learning a programming language, so the disc was occupied largely by short programmes and tiny data files. It was probably about the worst imaginable disc organisation for the task ! – but we didn't have the resources to try to improve it.

If the memory occupied by the disc map is still excessive, then it might be allowed to swap out in the ordinary virtual memory sense if it isn't being used. This is no more than saying that the map isn't part of the current working set; once it is required, it will be brought into memory, and then probably used so intensively that in any virtual memory system that takes account of usage it will be in no danger of disappearing again prematurely.

SPOOLING SPACE.

We can describe the structure of the disc area reserved for spooling as an **array[flex] of frozen stream**. The characteristic feature of files which are candidates for storage in this area is that they are in transit. Generally, they have been produced by one programme in order to be used by another. They might outlive the programmes which produced them, so the system must keep information about their whereabouts, origins, and properties, as no other entity is available to preserve it. While files on their way to or from devices such as readers and printers are the traditional residents of these areas, other sorts qualify; electronic mail is a prime example. Having said that, we must admit that we know of only one system in which such rational planning can be seen – most mail systems are separate packages, not integrated with the operating system – but the principle is sound enough.

The exception is IBM's virtual machine system, in which the metaphor of separate machines for every person using the computer is – or was – applied fairly consistently. To maintain this metaphor, the operation of transferring a file from one person's file system to another's required that a virtual card punch be used to convert the file into a virtual deck of virtual cards, which was then sent through the spool system to the recipient's virtual card reader, in exactly the same way as files from the real card reader. Though the metaphor had a certain olde worlde quaintness which took a little getting used to, the system itself had an undeniable elegance because of its basic simplicity and consistency. Unfortunately, it was also very slow, and the file transfer by virtual card punch was superseded by a much faster, though less interesting, direct file copy method called the Inter-User Communications Vehicle (or, more commonly, and neither more nor less comprehensibly, IUCV), which was quite distinct from the spooling system and made no use of the disc spool area. (We shall meet IUCV and virtual card files again in the chapter MAKING ARCHIVES WORK.)

Returning to the spool system and the information required to make it work, we observe that some sort of **directory** will be necessary to store the information. The minimum requirement is for the source and destination of the files, and, of course, where to find them. Additional information might include the file sizes, times and dates, and

perhaps some information about the sort of file if any significant variety is possible (as in the IBM system described above).

So far as **storage management** is concerned, files in the spooling space are invariably frozen streams, and usually of unpredictable length when begun; a sensible organisation is as a chain of blocks. (This is the first of the two storage methods described in the previous chapter. Though not well suited to general file systems, it is particularly appropriate in this context as there is no requirement for random access, and the method deals with files of arbitrary length with very little administrative overhead.)

The basic **operations** provided are those required to accept records from a source – usually someone's programme – and collect them together into a file. There should be no need for the programme to know that its destination is a spool system rather than a device, so procedures to handle all the sensible file operations – or at least serial file operations – must be provided. Apart from that, and corresponding procedures for moving the files out of the spooling space, few spool systems allow any access to their contents.

This is unsatisfactory. Most people have had the experience of sending a file for printing, and almost immediately realising that it was a mistake. On most systems, it is impossible to recall the file, even if it is still lurking in a spool queue. It should at least be possible to cancel a file in spool space, subject to proper security constraints, up to the point of cancelling printing or other transmission even if it has already begun. The IBM system we have mentioned offered unusually open access to the queues; one could inspect the queue associated with any virtual device, delete files, reorder them, assign priorities, transfer to other devices, and so on. It is only fair to add that there was no concession to inexpert users; you had to know a good deal about the system to make it all work well, but at least the operations were possible.

Our alert readers will naturally have noticed that we are trying to have it both ways. After emphasising the merits of a simple system metaphor, we are now complaining about it. If we use a real printer, then once the printing programme has finished there is no way to unprint the text from the paper. For a simple system metaphor, the same should be true of a system which happens to use a spooler. Certainly having to know that there exist spool queues somewhere is a complication. How, then, can we justify our complaints ?

We can justify them by pointing out that this is not a manual for people using an operating system, but notes for people studying operating systems. Someone has to decide what the system illusion shall be, and to do so it is necessary to decide what facilities are required, and then to choose the metaphor to fit. If spool facilities are desired, then perhaps the appropriate metaphor for files to be printed isn't a printer but something like an "out" basket. You can retrieve documents from an "out" basket for modification until they are collected, typically at some unpredictable time in the future. Now a real printer becomes an extraordinarily, but not impossibly, prompt postal service.

SWAPPING SPACE, FOR VIRTUAL MEMORY.

The structure of the swapping space is the simplest of the lot :

(for pages) an **array[N] of page**;

(for segments) an **array[flex] of array[flex] of byte** .

The reason for this simplicity is that the swapping software of the memory management system, in conjunction with the processes' addressing tables, deals with almost all the administration without involving the disc end of the operation more than absolutely necessary. This is clearly sensible, as it is exceedingly important that virtual memory should not be hampered by any administrative overhead beyond what is absolutely unavoidable. The addressing tables therefore contain all necessary details, such as disc addresses, ready for immediate access when required, and there is therefore no need for a **directory**.

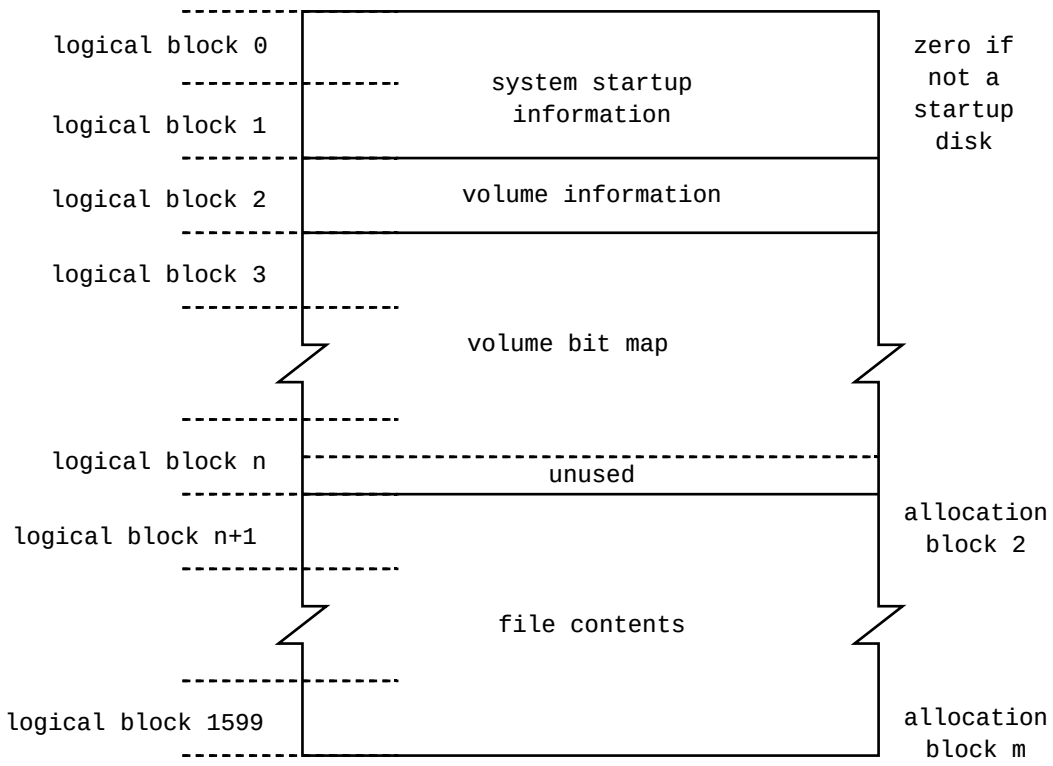
Storage management is restricted to a record of vacant and available disc space. The system only *needs* to know which areas of the space are in use and which are free, so that it can make allocations on request. All other information about what's where is in the processes' memory tables, so there's no evident structure in the swapping space itself. The only **operations** which must be provided are those which allocate and retrieve space in the swapping area – once again, everything else is handled by the virtual memory software.

SYSTEM MATERIAL.

This material is various in nature, including information on the structure of the disc storage itself (the identity of the disc, where the different sorts of data are kept, pointers to special things such as bad sectors, amount of available space, and so on), and system code. Because of this, the only safe description of the structure is a **frozen stream of byte**. (It's true that there's structure in the code, but that isn't visible to the memory and disc.)

An important component under this heading is the initial memory load used for starting the computer system. This is typically recorded as a simple byte stream to be read into memory when the machine is restarted. Old systems used to have a lot of this, but more recent ones can save much of the system in ROM. It is typical of the early remarks on disc usage that the disc space required for the system has gone up despite this apparent economy feature. It is noteworthy that, following the trend towards operating system microkernels, much system material is now commonly kept in the ordinary file system rather than being stored in a special way.

Even so, there is material – parameters, configuration details, and such – which cannot be fixed permanently, and must therefore be saved on some rewritable medium. The Macintosh disc layout[SUP4] is an interesting example :



The disc map in the diagram is drawn for the older, 800 kilobyte, discs, but the pattern is designed to be extensible, and works with modern large discs as well. A *logical block* is the equivalent of a disc sector – 512 bytes of data, and 12 additional bytes of "file tags" from which the disc structure can be rebuilt if the directories are lost. The volume bit map contains one bit for each *allocation block* of the disc; an allocation block is the group of sectors which we have called a cluster, and its size is the same throughout the disc.

COMPARE :

Lane and Mooney[INT3] : Chapter 12; Silberschatz and Galvin[INT4] : Chapter 11.

REFERENCES.

SUP4 : *Inside Macintosh*, volume 4 (Addison-Wesley, 1985).

QUESTIONS.

What's wrong with the first two mentioned ways of keeping track of the vacant space on the disc ? Does the disc map overcome the defects ?

What provision must the spooling system make to collect files record by record and build them up into a complete file ? Consider the information which must be kept in the directory, and structures in the file system itself.

What are the "proper security constraints" required to guarantee the safety of the additional spooling operations suggested in the text above ?

How would you build a better file system for the Tops-10 system running as described in the example ? Take into consideration that the file headers and trailers were used to implement links between the clusters which belonged to a file, and to make it possible to reconstruct the file directories if they were lost or corrupted – which they not infrequently were, usually by system

failures in which the current directories were not written to the disc from memory.