REAL-TIME DISC SYSTEMS.

Over the past two decades, the speed of the electronic circuitry used in processors has increased phenomenally, and this change has had two significant consequences for computers' disc systems.

First, the increase in speed has brought with it a requirement for more disc space, and disc technology has advanced to provide much higher storage densities and much cheaper disc units. The portable computer on which this text is being typed has a 350-megabyte disc, which probably exceeds the on-line storage available in the whole university in 1980. Because of the low cost, organisations can install many disc units and achieve very large data stores – but that policy brings back reliability problems which we thought had vanished into the past. Modern individual disc units are very reliable, with expected mean times before failure of the order of fifty years, but, assuming a constant probability of failure (which is somewhat pessimistic, but it's better to play safe), that implies an expected mean time before failure for an installation of a hundred discs of six months.

Second, to support the increase in processor speed faster disc service is necessary. This relationship is not as obvious as it appears at first glance, for – as we have seen – a significant part of the processor power has been used to improve the user interface and the general availability of good services rather than supporting more "real computing", and the reduction in price of memory which has accompanied the developments in circuit technology has led to much larger memories, which reduces the demand for fast disc access. What has happened instead is that the faster processors have made it possible to offer services which were impracticable with older machines, such as displaying real-time moving pictures, which can consume data at rates of up to several megabytes per second for comparatively long periods.

These requirements introduce two sorts of problem, which we shall approach in our favoured top-down direction. First, we must analyse the requirements to find out just what we will have to do to satisfy them; and, second, we must find ways to implement the design. Our interest in both cases is to find out what demands will be made on the operating system, and to devise means of satisfying those demands – as always, the operating system must provide such services as are needed to make it possible to write programmes effectively.

AN EXAMPLE.

We begin with a realistic example of a rather simple multimedia presentation of the sort which have fairly recently become common. Consider this specification[SUP7] :

> A simple slide-tape presentation illustrates how some applications must explicitly anticipate storage latencies. Say a digital slide-tape presentation requires that slide image *n* from a database be displayed within 0.1 second of synchronization audio event *n* in a sound track, so that a viewer perceives the two events as simultaneous. Once the sound-track presentation begins, it cannot pause. Assume that each stored image consumes 1 megabyte, the presentation consists of 100 images, and the synchronization events occur 1 to 10 seconds apart. The platform consists of a workstation with 16 megabytes of memory, and the stored images are distributed across several platters of an optical disk jukebox. A realistic seek time for current optical disk technology is 0.5 second; disk replacement in a jukebox can take 4 seconds. Clearly, it is not feasible to wait for a synchronization event before attempting to retrieve the corresponding image.

The major problem is to ensure that the next image is ready in memory when the request to display it happens. (In this discussion, we'll ignore the sound stream and any data compression and expansion; in practice, both of these must be taken into account, but the pictures dominate the problem.) The time to retrieve an arbitrary image from the disc system might be up to four seconds; the image must be ready within 0.1 second.

We know how to do that : we need buffers. That's our standard technique for coping with mismatches between device speeds and processor requirements, and it works here just as well as it did with a tape-to-tape payroll system in 1950. Apart from matters of scale, the main difference in principle is that we don't know as much about the pattern of requests, and that, combined with the real-time constraints, is the special feature of the multimedia system.

The worst case is a stream of synchronisation events at the minimum spacing of 1 second; we should be able to cope with that reasonably well unless the sequence requires several disc replacements, but there isn't much time to waste. For the reasons we mentioned when describing disc scheduling, it is best to present several disc requests at once so that the disc controller has a chance of optimising the request sequence; generally, we want to make requests as far in advance as possible, given the buffer space available.

From that description, two factors stand out as important in determining the quality of service which the system can offer. Both are to do with the predictability of the system, which we want so that the system can plan its disc operations ahead to give a better chance of completing them. We'll describe the two variants separately, but in practice they are fairly tightly linked.

PREDICTABILITY IN TIME : how much information is needed to predict request times ? We would like to know this in order to decide what to do next, and how urgent it is. This is particularly important in a shared system, where requests from different sources must be satisfied by the same disc installation. The answer to this question depends to a great extent on the nature of the process and how it is controlled. Continuing with the multimedia example, a real-time video presentation has high demands for data, but they are completely predictable – the system requires 25 (or so) pictures per second at a regular rate until the end of the file, or further notice. At the other extreme, a system controlled interactively where people can display single pictures or movies at any time is unpredictable. Here's an attempt at classification; it is by no means exhaustive, but it serves to illustrate the important points :

Nature	*Characteristics*	*Programme type*

PERIODIC	Regular requests	A single continuous stream.
PREDICTABLE	Irregular, but known beforehand.	Preprogrammed selections.
UNPREDICTABLE	Not known beforehand.	Interactive control.

For *periodic* requests, the system knows from the start that there will be a regular requirement for service at some known frequency for a known time to come, and can set aside priority scheduling slots to handle these requests and work out sensible buffer allocations. These are usually urgent, as any failure will hold up the process in an activity intended to be continuous. With *predictable* requests, the system knows that there will be requests from some process for data from a specified stream, with predefined but irregular time intervals between them. The slide-tape example falls into this category. The system can plan ahead, reserving time in advance to satisfy the requests. Again, buffer requirements can be determined. An *unpredictable* request stream is – reasonably enough – unpredictable, and the system can do very little but allocate one or two buffers. Notice that requests must be at least predictable if the system is to guarantee performance.

PREDICTABILITY IN SPACE : how is the position of the next item defined ? This is an important question, primarily at the disc scheduling level, for requests scattered randomly over the available storage medium are much harder to handle than the same number of requests arranged tidily in a serial file. There are also side effects on the temporal behaviour, for if it is not known beforehand whether or not a change of disc will be required to satisfy a future request it is necessary to (try to) deal with the worst case, and to allow enough time to change the disc. Once again, we can distinguish three levels of organisation :

Nature	*Characteristics*	*Programme type*
SEQUENTIAL	Get the next record.	A single continuous stream.
SCRIPTED	Defined by a fixed programme.	Preprogrammed selections. (Allows for loops, etc.)
DYNAMIC	Computed when required.	Interactive control.

Sequential requests are simple and orderly. *Scripted* requests might require more planning ahead, but provided that the sequence is known planning is, in principle, possible. For *dynamic* requests, little can be done; without forewarning of the location of a required item, no preparation is possible, and service cannot be guaranteed above some minimum level.

Summarising these comments, it is clear that the important factors in determining the level of service which can be offered are :

LATENCY : how long it takes to get data from the storage medium;
BUFFER SIZE : the limit on how much can be saved in memory; and
STATISTICS : how one stream of requests can be interleaved with others.

These factors are available to, or under the control of, the operating system, and, given sufficient information about the processes' requirements, the system can – in principle – determine whether the requirements can be satisfied, and, if so, how.

OPERATING SYSTEM RESPONSIBILITIES.

To what extent should the operating system be involved in these activities ? The simplest answer is that it need not be involved at all. We can write our operating system, but decline to offer any real-time file services on the grounds that they are a highly specialised requirement, and we can't afford to provide the necessary facilities in our general-purpose operating system just to keep a very few people happy. (Well, it sounds better than saying we're too lazy.)

That's a perfectly good answer, provided that the requirement remains highly specialised. In fact, that isn't the way that matters are going. While multimedia systems are perhaps the most spectacular examples – though they are by no means as important or significant as their proponents would have us believe – there are other good reasons for an increasing interest in file systems with guaranteed real-time performance. They are valuable in any system in which large amounts of data are moved about at high speed, or in which very fast response to requests is important. We conclude that an operating system is more and more likely to be required to offer a real-time file service; what should this service look like ?

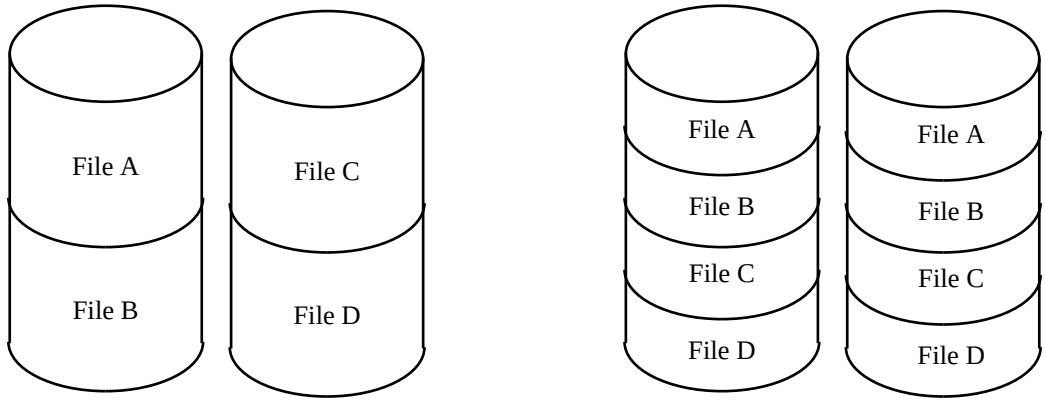
The simplest requirement is for supervisor calls corresponding to the conventional file-handling calls, but with provision for the specification of the required response time; the system will either accept such a request and perform it, or return very quickly with a result descriptor implying that the operation is impossible. At a higher level, some means to give the system prior information about predictable requests will be necessary, but we know of no current system in which these requests are available.

The methods available for achieving the desired performance, in addition to those currently used in disc scheduling, are of two sorts. The first is implied by our discussion of predictability; *prefetching* can be used to get required data into memory before they are needed, given the best knowledge available of the programme's behaviour. The second looks ahead even further. For any predictable programme, *presequencing* can be used to overcome problems of wide spatial distribution of material. In effect, the programme is precompiled, constructing a sequential file of all the information which will be needed. Whether it is sensible to incorporate presequencing in an operating system remains to be seen; perhaps a more obvious approach would be to provide presequencing facilities in software used to construct real-time programmes.

SPEED, RELIABILITY, AND ARRAYS OF DISCS.

It seems, then, that, most of the time, we can design systems to achieve the performance we need, provided that we can manage to get raw data quickly enough. The principle is to use a lot of discs, and to combine them together in some appropriate manner. Generally, reliability and speed remain as problems to be solved. Most unusually, there is a single solution to both problems.

First consider the speed problem[IMP28]. Earlier we mentioned and installation of a hundred disc drives; each drive might be able to supply data at, say, one megabyte per second. (You can do better than that, but it's a nice round number.) Suppose that your display requires data at the rate of about ten megabytes per second. (Considering that we need something like a megabyte for each frame for a reasonable black-and-white picture, and 25 frames per second to make movement look convincing, that sounds low, but we can manage data compression well enough to make it realistic.) We can get the data rate we need by using ten discs in parallel – but that will only work if we arrange the data in a rather unusual way. Instead of, as is conventional, storing the data for the file on one disc, we have to spread the data for the required file over all ten discs. This is called *striping*, for reasons which may be clear from the diagram below.



How thick are the stripes ? – which is to say, how far do we have to destroy the original file sequence to make the system work as we want ? The answer depends on the amount of memory we can use for buffering. Consider two examples.

First, suppose we take striping to the extreme with stripes one byte in width. Then one stripe across our ten discs will give us altogether ten sequential bytes from the file. It's unlikely to be possible to control the discs with sufficient precision to get these at *exactly* the right moments, so we'll allow for a double ten-byte buffer, which will let us fill one buffer while the other is emptying. That would work in principle, but would be hard to manage in practice. We'd have trouble at disc sector boundaries, track boundaries, picture frame boundaries, and so on. (We ignore an important reason for requiring bigger buffers, which is the need for a reasonably large volume of data to decode, because this isn't a course on data compression. Just take our word for it.)

Suppose, then, that the stripes are one frame wide, so that we read a frame from each disc, and successive frames from successive discs. Each disc can fill its own megabyte buffer in one second, but in that time we will require ten megabytes to work on and decode. If we use double buffering again, we'll want 20 megabytes of buffer; but that's enough to smooth out the fluctuations in disc data rates and to provide lots of material for the decoding.

It's clear that there is a relationship between the data rate required, the thickness of the stripe, and the size of the buffer used. We shall not work out the details here, but the important point is that it all works. That solves the speed problem.

Now for the reliability problem. We commented earlier (in this chapter, and in *DISCS – THE HARDWARE VIEW*) that using many discs necessarily increases the chance of failure, and we must take precautions against this if we are to construct a reliable system.

We can do this by using error-correcting codes[IMP29]. These work by storing additional data computed from the real data in such a way that any error in a certain data string can, provided that some limit on the number of errors is not exceeded, be corrected from what is left of the string. In a multiple-disc system, we can store the additional data as a stripe on a separate disc, read the redundant stripe in parallel with the data with which it is associated, and then perform the error correction computation. The number of additional streams necessary depends on the level of protection required, but an increase of a few percent in the data volume can give a very useful degree of protection.

A system of this sort is commonly called a RAID, which stands for Reliable Array of Independent Discs, or Redundant Assembly of Inexpensive Discs, or almost any combination of the two, and maybe a few more. Whatever the precise meaning of the acronym, the idea is clear : we use a lot of discs, and provide means of coping with faults. The second interpretation introduces a new theme; because of the redundancy, we can in fact use comparatively inexpensive disc hardware, which is an important point if you want a hundred of them or more.

REMARK.

We've used the requirement for very fast data transfer to make the case for RAID methods, but it can be justified purely on reliability grounds. As it happens, both requirements have become pressing at about the same time, and it is fortunate that this technique can meet both requirements.

REFERENCES.

SUP7 : R. Staehli, J. Walpole : "Constrained latency storage access", *IEEE Computer* **26#3**, 44 (March, 1993).

IMP28 : G.R. Ganger, B.L. Worthington, R.Y. Hou, Y.N. Patt : "Disk arrays high-performance, high-reliability storage subsystems", *IEEE Computer* **27#3**, 30 (March, 1994).

IMP29 : G.A. Gibson, L. Hellerstein, R.M. Karp, R.H. Katz, D.A. Patterson : "Failure correction techniques for large disk arrays", Proceedings of the third international conference on architectural support for programming languages and operating systems, *Operating Systems Review* **23** Special Issue, 123 (April 1989).

QUESTIONS

What sort of disc directory would you want for a striped system ?

If one of the discs fails, it can be reconstituted from the others. How could the system carry out the reconstitution automatically ?
