DEVICES

As well as looking after everything that happens inside the computer processor and memory, the operating system has to administer everything outside – or, at any rate, the parts of outside which are attached to the computer. They turn up as the sources of and destinations for streams; they are the entities, which we shall call *devices*, at the other ends of the stream operations we met earlier. We've already met some of these devices : terminals and discs have been prominent, and others have fleetingly appeared, but all this discussion has been at a fairly high level, and we haven't worried a lot about how it is all to be implemented. On the other hand, we *do* have some idea of what we want to implement, and why.

As we have previously noticed in our discussions of memory and processes, one of the interesting consequences of getting closer to real machinery (which had to come some time) is that we can no longer blithely organise things as we would like them to be; we have to grapple with what actually is. We remark that our preoccupation with the abstract does not derive from some Utopian rejection of reality, but from the need for an orderly approach to a disorderly world. The chemist does not start a systematic study of matter by peering at copper sulphate and benzene (or whatever they call them nowadays), but with ideas of atomic theory and energy levels, and we have tried to adopt an analogous approach. At the same time, the chemical theories are no use unless they can be applied to copper sulphate and benzene, and we are now approaching the corresponding point in our study of operating systems. There is nevertheless still good reason to try to preserve what order we can for as long as possible. First, it is far easier to organise any collection of things if we can treat them as parts of some structure, with common features and known relationships with each other. Second, we saw when discussing *FILES IN THE SYSTEM* that *device independence* was a worthy goal; this too will be easier to approach if we can treat things in a similar way for as long as possible. So we'll proceed in four stages.

- First, we'll follow the idea of common structure and device independence to introduce a structure called the device table which we'll need quite frequently later.

- Second, we'll continue to deal with streams generally, but concentrate on some common implementation issues.

- Third, we'll deal with two special cases, both instances of devices which are not used in a simple way, and which therefore need another layer of administration before we get down to the hardware. In both cases, that's because the devices must cope with many streams, not all of the same sort, simultaneously. *Terminals* can be used by programmes as ordinary input and output devices, but they also act as our control devices, and provide our interface with the operating system. *Discs* are used for many different sorts of storage : we've mentioned ordinary files, spooling, and virtual memory, and they're also used by the system for various tasks.

- Fourth, we'll discuss how the operating system deals with the hardware devices proper.

THE DEVICE TABLE.

Perhaps the most difficult step in developing an orderly treatment of devices in the operating system was to find a reasonably simple representation of devices which would cater for all types of device, with their enormous range of structures, forms, and requirements. What sort of orderly structure will cater equally well for a mouse, a keyboard, a screen, a printer, a pair of loudspeakers, a modem, and a disc drive ? (And that's just staying with conventional systems. Add robot, dataglove, virtual reality headset, etc. if you like.)

The answer is somewhere between "there isn't one" and "that's the wrong question". There isn't one, because at the obvious level the devices are indeed much too

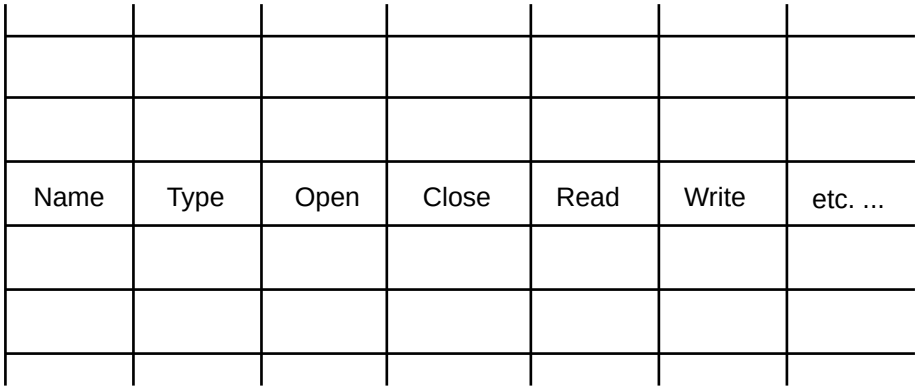
diverse to be forced into the same pattern. In all the other tables we've used – userdata, file, process, and so on – we've known just what we wanted to do with each entity in the table, and they've all been much the same. Devices are quite likely to be quite different, and we really don't know what we might want to do with the next one because it hasn't been invented yet.

And that's why it's the wrong question. The right question must be something we can answer. It was because we couldn't answer the obvious question that it was some time before devices were handled by operating systems in anything resembling an orderly way. We would like to conjecture that the cause of the delay was the traditional bottom-up view of operating systems which guided (if that's the word we want) their development; our evidence is that, with the approach we've used in this course, we have already laid the basis for a question which we *can* answer. This came in *STREAMS IN PROGRAMMES*, where we were able to discuss a programme's interaction with the outside world in quite general terms even though we hadn't said anything at all about the peculiarities of the devices. In these terms, a more appropriate question is "What is the level of generality at which we can build an orderly structure to manage devices ?" – and a possible answer lies in our pattern of control and data streams which we developed in that chapter.

Following that argument, we might expect that some structure which could provide access to means of controlling input and output of data and administrative information would go some way to providing the facilities we want. Of course, the implementation details of the various control methods would differ greatly from device to device, but the uniform means of access should give the regularity we need to construct our system.

It turns out that this really works, and the result is some sort of *device table*. The device table is composed of an entry called a *device descriptor* (the Lane and Mooney textbook[INT3] gives five other names) for each device known to the system. All the device descriptors are the same in structure, and they contain, directly or indirectly, anything the system needs to know about all devices, such as information about the device state, what the device is doing at the moment, which process owns it (which only makes sense for a device that only does one thing at once), where to find the software which manages the device (the specific procedures for file operations, which we met in the earlier discussion, and the lower-level device driver and interrupt handler, discussed further shortly), and so on. The device status information must be changed as required by whatever software detects the change.

Here's a diagram to illustrate the principle. This is intended to demonstrate the idea, and isn't – so far as we know – the way the table is implemented in any particular system. (For one thing, we've missed out a lot which we'll deal with in this section.) Also, as usual, different operating systems have notably different ways of dealing with the devices which they control. The diagram assumes that the descriptors are collected together into a table rather than linked, and that the input-output request blocks (see below) are linked rather than collected; it makes little difference in practice.



We suppose that each device has some sort of name, so that you can say which you want, a type to distinguish a printer from a keyboard, and a set of pointers to procedures for managing the various data movement and control operations. A real device table is likely to have much more than this, but this illustrates the basic structure.

The device table is initially established when the system is started up. At this time, a device descriptor must be constructed for each device, and added to the table. This is called *installing* the device. The neatest way to manage the installation is to have an installing programme for each device. This loads the device software into memory, does whatever must be done to direct the device's interrupts to the right place (the interrupt handler), constructs the device descriptor in the device table, and does any other necessary housekeeping, such as making sure that the device is there and testing it if required.

With a really clever system, you should be able to install or remove devices without stopping the system. (That's roughly what Microsoft call "Plug and play"; it is worth recording that the same feature worked with Burroughs systems in the early 1970s.) There is nothing particularly difficult about this, but it does require that the system be designed to identify devices by name rather than by memory address or position in the table; clearly, such low-level "names" can't be guaranteed if you don't know beforehand just what devices will be available at any time. That's the principle of late binding again.

IMPLEMENTING STREAMS.

In our earlier treatment of streams, we decided that we needed certain services from the operating system : ways to use streams in programmes, ways to use names to find files, ways to handle stream devices, and somewhere to put files. Here we'll look further into using streams in programmes, and the other topics will follow in later sections.

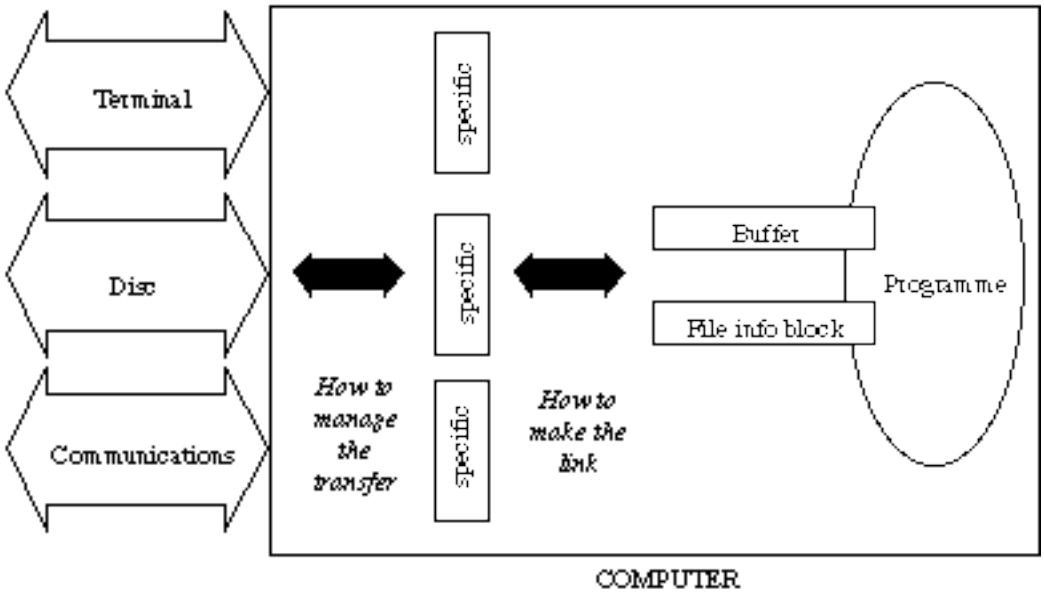
The interface we have to provide was described, if not precisely defined, when we discussed *STREAMS IN PROGRAMMES*. It has two components :

- a *file information block* for the administrative requirements; and
- a *file buffer* for moving data between process and stream.

We identified a number of operations which were necessary to handle the streams in the ways which we thought appropriate. Now we shall follow these through to a lower level of detail. There is a lot of detail to cover; some examples are :

- *data transfer* and *administration*, corresponding to the two components just mentioned;
- operations required *on every transfer* or *once only*;
- *input* and *output* transactions;
- operations which are *general* or *specific to some device*.

Generally, we shall be discussing the two topics we need to fill in the gaps left in our earlier discussion at a higher level; they are shown on the diagram below as black two-way arrows.

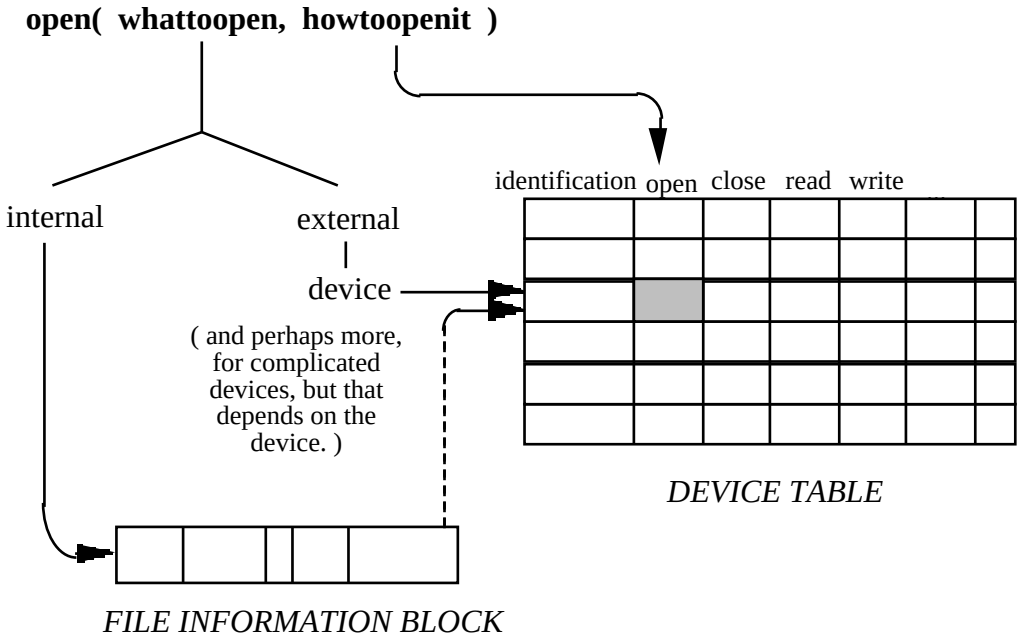


OPENING AND CLOSING.

These operations are particularly connected with the *file information block.* This should contain all the information which the system needs as it works with the stream in normal operation. This information must come partly from the programme, and partly from outside – from a stream device, or from the file system. The **open** instruction gathers this material together, and thereby prepares the stream for use. Different languages use different syntactic representations, but the general form of the instruction is

open(whattoopen, howtoopenit).

whattoopen is enough information to identify the stream, both to the programme and to the operating system. The internal name used in the programme to identify the stream selects the file information block to be used; further details must be provided with which the system can construct or retrieve all required information about the stream in the external environment. This will usually require some device identifier, and perhaps other particulars, depending on the device, though this information need not always be given explicitly. In a capability system, the other particulars would include a capability for the stream. Here's a diagram[IMP32] to illustrate how it all fits together :



The **open** operation uses the *external* part of the device information to construct the link (shown as a broken line) between the file information block (identified by

the *internal* part of the device information) and the correct device descriptor in the device table. Once the link is established, we don't need the external name any more. Having identified the descriptor, we can now perform operations on the device itself – so the **open** operation might well continue by executing the specific open procedure identified in the device table as that to be used for this device.

For devices which can handle only one stream at a time – printers, tapes, etc. – the device identifier is in principle sufficient identification, but isn't necessarily the most satisfactory choice. In the case of removable storage media (magnetic tapes and discs, for example), merely defining the device could be disastrous if the wrong disc or tape is used. It is therefore common practice to give every such item a standard identification area which is easily accessible (at the head of the tape, or in a standard disc sector) and can be checked before the storage medium is used. **whattoopen** in this case should identify the device *and* the medium volume to be used.

It was common practice at one time to provide files in general with standard headers. A file header is a prefix attached to the file proper, containing information on the identification and nature of the file; using this information, the system can check that the correct file is indeed where it's expected. The header was particularly useful for tape files, especially when tapes were moved from site to site. They would have been even more useful if they had been really standard.

For more complex devices which can handle several streams simultaneously – traditionally, discs, though terminals with windowed displays now qualify too – more information is needed. It is usually easy enough to define what information is needed, but not nearly so easy to decide how to give it. For a disc file, we must identify the disc and give the address of the file on the disc, but it is quite unusual to present this information explicitly. Instead, we present some information from which the system works out the significant details – so we may give the name of a disc and the complete file name, leaving it to the operating system to find the file in the specified disc's directory, or we may simply give a local file name, when the system must build up the complete file name from our search profile, and determine the disc name from the directory structure somehow.

In some operating systems – Unix is an example – the distinction between different sorts of device is hidden by including all files and streams in the same directory. (A terminal in Unix is disguised to appear as just another file with a name like /dev/tty.) To make this work, all the files and streams must be in the directory somewhere, and all the other information required by the system to manage the files must be kept in the directory. The directory no longer corresponds to a single disc or other device, and it must be possible to attach the local directories of new devices to the full directory as branches of the tree structure. This is known as *mounting* the new directory or device.

howtoopenit might not be required with some simple devices which can only work in one way, but it might be necessary to say whether the stream is to be read or written, to specify protection conditions which must apply, or to give some other information about structures which must be established – what size buffers, how many buffers, (for a file) what sort of file access technique should be used (sequential, random, indexed, etc.), and so on.

We have sidestepped a number of problems in language design with our bland **open( )** instruction. The internal structures of both **whattoopen** and **howtoopenit** might be very different for different sorts of stream; how can these all be assimilated into the same syntactic form ? Bear in mind as you contemplate that question that your proposed

method must be able to cope with streams associated with devices which have not yet been invented. It is hardly surprising that some designers have given up and required that the descriptions be presented as strings, to be passed on uninspected and unchanged to the device software which is intended to understand them. An alternative is to leave space in the syntactic definition for a general data structure, which can be different for different devices. The flaw in these catch-all approaches is that the compiler does not see the information presented, so errors cannot be caught until the programme is executed. On the other hand, even if the compiler could see the detail, how would it understand the meaning ? While it isn't impossible in principle to ensure that a compiler could identify such errors it would be necessary to provide some means for the compiler to find out about all the different devices, some of which might not exist when the compiler is written, and we know of no system in which this theoretical ability has been implemented.

The result of executing **open( )** is an open stream, and a corresponding file information block. The information contained in the file information block is determined by the system's requirements for handling the stream, but is likely to include something like this :

Identification :

Device : Points to the system's representation of the device in the device table.

Owner : Identifies the process which owns the stream.

More details, if needed : For a simple device, perhaps null; for something more complicated, information which identifies the intended stream within the device. For a disc, a filename; for a graphics screen, a window identifier.

Structures in memory :

Buffer pointer : Where the buffer is – or where they are, for a multiply buffered stream.

Buffer length : Guards against overflow.

Current state :

Stream state : open, closed, reading, writing, at end, ...

Current character pointer : Next character position in the buffer.

Current record number : In case anyone wants to know.

It should be stated that it's fairly unusual to find all that information collected together quite so neatly – but most of it must be there somewhere, so even if there is no explicit file information block the principle is still sound.

When we have finished with a stream, it must be **close**d. Some compilers automatically ensure that all streams are closed when a process finishes, but with others you have to close the streams explicitly. Neither of these is much use if the process doesn't finish normally, which is where the **owner** field comes in : if a process stops abnormally for any reason, the operating system can inspect its file information blocks, identify the streams which the process was using, and close them tidily. That's another good reason for keeping the file information block in the system area, where it is less likely to be destroyed if things go wrong. Details specific to the device are kept in the **close** procedure in the device table.

However it is managed, the aim of the operation is to tidy up behind the stream. It will be clear from our discussion that there are two parts to this : cleaning up the representation of the stream in memory, and making sure that the external part of the stream is properly tidied too. For a real persistent file, this involves an additional task : it is necessary to ensure that the information kept in the permanent file directory and in any collections of information about the recording medium (such as a list of occupied areas on a disc) is consistent with the actual state of the file. In memory, the file information block for the file to be closed must be appropriately marked, and the buffer must be emptied and (perhaps, depending on how the system is organised) returned to the memory manager.

For a stream, details depend on the nature of the stream and the other partner, or partners, in the communication. For a simple device driven directly from the computer, nothing at all might be necessary. In other cases, some sort of end-of-transmission protocol must be observed.

To summarise, these are the steps involved in closing a stream :

- If the stream is open for output, any unwritten material in buffers must be copied to the stream.
- The buffers can then be disposed of.
- Any special operations required to restore the stream and device to a disengaged state must be carried out.
- The file information block must be amended to show that the stream is closed.

READING AND WRITING.

These operations are mainly concerned with the *buffer*, though they rely on information in the file information block, and might have side effects on its contents. We introduced buffers earlier, in the chapter *STREAMS IN PROGRAMMES*. They are almost always necessary because of mismatches between programme and hardware (or operating system routine) either in speed or amount of data transferred; for example, a programme requiring a block of 50 bytes of data each second might be reading from a disc which can only supply data in chunks of 1 kilobyte, but at a rate of 1 megabyte per second. Many devices cannot sensibly read and write single bytes or words of data; instead, they handle data in comparatively large chunks, and the buffer is used to assemble such chunks on output, or to hold chunks read on input until they are consumed. Even a device which can handle data in small quantities might require a little time to do so, and then it is convenient for a programme which uses a device of this sort for output to deposit the information in the buffer and continue, rather than to wait for the transaction to be completed. We are concerned with the interaction of the buffers in two directions :

Towards the programme : When a programme reads data, it expects to be able to see the next item from the stream; when it writes, it expects the item written to be moved to the stream. There are at least two ways to achieve this. We can either give the programme a pointer into the real device buffer, and advance the pointer for the **read** or **write** operation; or we can give the programme a second buffer, matched to its own requirements rather than the hardware's, and copy the information between the stream and programme buffers. Both methods have been used, but the second is much more common. (It's interesting, though, that the behaviour of the READ and WRITE instructions in Cobol was defined to suit the pointer-into-the-buffer method; after a WRITE, the contents of the output stream's declared record are undefined.) In both cases, of course, we need occasional exchanges of information between the buffer and the "real" stream.

Towards the stream : The physical transfers must be managed in the correct way at the right time. This is not necessarily simple, especially if the device can only transfer information in blocks. Consider the disc which can only transfer information in 1 kilobyte sectors, and a random access operation to write one byte. The operation can only be executed by fetching the whole sector containing the byte, changing the byte in the buffer, and writing the whole sector back again. Clearly, these details are the system's job, not the programmer's, so the device **read( )** and **write( )** procedures must be able to cope with them.

How is all this activity coordinated in practice ? We'll describe the structure in more detail later in *MAKING DEVICES WORK*, but here's a summary of the process to illustrate how the different components cooperate. As always, details are liable to variation between systems, but this is a plausible sequence of the operations involved in executing a **write** instruction, expressed in terms of the structures we've described. The sequence is initiated when the compiled version of the source programme's **write** instruction is executed. This is represented by a call (probably a supervisor call) to a standard procedure, with parameters representing a request for a write operation, a

specification of – probably a pointer to – the data to be written, and a link of some sort to the file information block set up for the stream to be used. The standard procedure (which we shall later call **doio**) refers to the file information block to identify the device, and then refers to the appropriate device table entry. It might then ensure that certain conditions are satisfied (for example, is the device available ?), after which it can execute the device's write procedure.

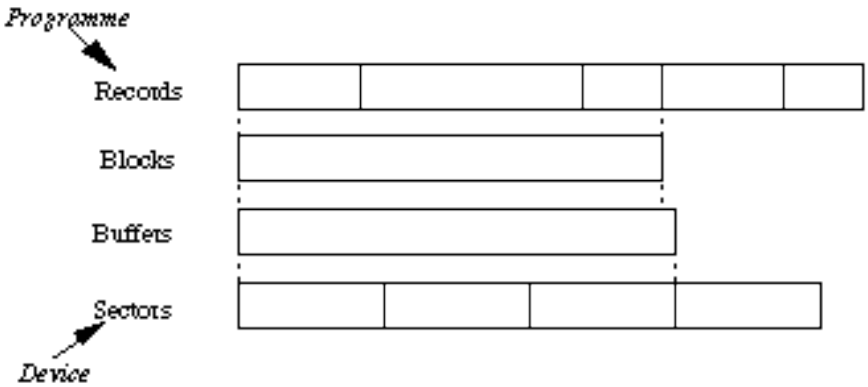
RECORDS, BUFFERS, AND BLOCKS.

Both the "Towards ..." paragraphs above illustrate circumstances in which what the programme wants isn't what the device provides. The relationship between the programme's input and output requirements and what the system can provide is essentially the same as that between segments and pages in memory management : one is determined by the programme, and varies from case to case, while the other is strongly dependent on, if not fully determined by, characteristics of the hardware.

The preoccupation with buffers which was evident in the earlier days of computing was partly concerned with speed, where it was often good to reduce the number of hardware operations to the minimum, and partly with optimum use of the storage medium, which was very expensive. Today the cost of the storage medium is rarely of overwhelming concern, but the demand for speed is still there – if anything, more insistent than ever before, with the requirement for real-time digital video presentations, perhaps synchronised with several other streams of information for multimedia work.

The hardware constraints are often concerned with data being transferred only in chunks. Sometimes that's because of the physical nature or organisation of the medium : all disc sectors are the same size, and to read or write a sector you have to start at the beginning. In other cases, such as magnetic tape or communications lines, it is inefficient or impossible to transmit information in small chunks, so we have to transfer larger quantities.

It's therefore quite common to choose a chunk of data which is a, perhaps approximate, lowest common denominator* of both the programme's required units (which we shall call *records*) and the hardware's units. This unit of transmission is called a *block*, and the practice of grouping data in this way is called *blocking.* This diagram illustrates the relationship :



Notice how the sizes of the block and the buffer are chosen to match the requirements of the programme to the abilities of the hardware.

Notice that this blocking has no direct connection with the sort of blocking which happens to processes when they cannot proceed. It's just unfortunate that the textbook written by Lane and Mooney[INT3] has a

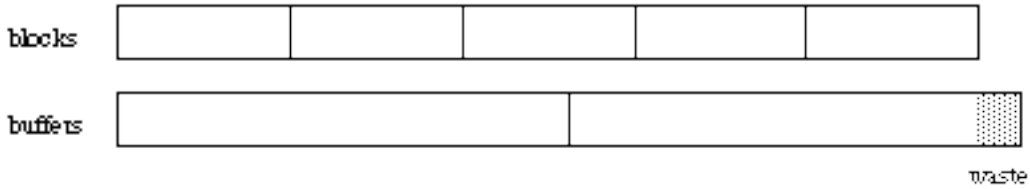
---

* - And we do mean "lowest common denominator". The term is often used to mean some level beneath those of a set of contributing parts - "Individual Members of Parliament behave reasonably, but when they meet their group behaviour sinks to the lowest common denominator". In fact, the lowest common denominator of a set of numbers is the smallest number of which they are all proper divisors, so is at least as big as any and usually greater than most.

section (page 224) entitled "Buffering and blocking",
which discusses something like this sort of buffering
and the other sort of blocking !

Blocking assumes a particularly simple form if both the records and the hardware-defined units of transfer are of fixed sizes, as is common with fixed-length record files which live on discs. Provided that the file is used serially, it might be possible to achieve significant gains both in data transmission speed and file space by choosing file blocks which fit both the programme's records and the disc's sectors reasonably well. For example, suppose that the disc sector is 256 bytes long, and the file record is 100 bytes long; with two records to a block, we waste about 20% of the disc space and it costs half a disc read per record :

If we store the file with five records to a block, we can use a 512-byte buffer, read two sectors sequentially at little more cost than a single sector, and in the process fetch five records and waste not much more than 2% of the disc space :



MULTIPLE BUFFERING.

Another way to use buffers to increase efficiency is to use more than one to permit parallel operations. There are several variants of the method, but they all rely on the principle of allowing different things to happen to different buffers. The classical example, still useful for simple streams such as those associated with sequential files, is to use two buffers. Once the system is under way, the idea is to work on one buffer while filling (or writing) the other, and to interchange between the buffers as appropriate. An obvious extension is to rotate through three buffers, with one filling, one working, and one writing. The simple pattern is conveniently illustrated for the special case of disc files, and goes like this :

With SEQUENTIAL ACCESS –

One buffer : processing waits while the buffer is filled or emptied. Here is a possible sequence of operations for a programme which repeatedly reads a record and performs some computation on it.

Fill buffer with record 1
Use record 1 in buffer
Fill buffer with record 2
Use record 2 in buffer
Fill buffer with record 3
Use record 3 in buffer

Two buffers : overlap processing with reading or writing. Here is the same programme executed with double buffering and two processes, one reading the stream and one performing the calculations :

Fill buffer A with record 1	

Fill buffer B with record 2	Use record 1 in buffer A
Fill buffer A with record 3	Use record 2 in buffer B
	Use record 3 in buffer A

This is clearly something of an idealisation, as it is most unlikely that the two parts of the overall operation will take exactly the same time, but in any case essentially all of the time taken for the faster part can be saved.

Three buffers : overlap reading, processing, writing.

Fill buffer A with record 1		
Fill buffer B with record 2	Use record 1 in buffer A	
Fill buffer C with record 3	Use record 2 in buffer B	Write record 1 from buffer A
	Use record 3 in buffer C	Write record 2 from buffer B
		Write record 3 from buffer C

This method is effective for tasks in which a complete file is updated to produce a new version. It only works under certain circumstances : you can only overlap reading and writing with a three-buffer system if the input and output files can be managed independently, which usually means separate disc drives.

With RANDOM ACCESS –

This is not quite so straightforward, as by definition we no longer know which record to fetch next until it's time to fetch it. We can therefore not manage any overlapping for read operations; but we could still hope to overlap writing with computing the next record. Blocking also causes trouble; for random write access to a small part of a disc sector it might be necessary to read the initial contents of the sector first, change the part which is written, then write back the changed sector.

It is also possible that by holding records in buffers we might find that a record we want is already present in memory, so we don't need to read it. This technique will *only* work with random access files (unless we can bring complete files into memory, which we sometimes can nowadays), but to make it effective we need as many buffers as we can get. In practice, this isn't sensible for individual files, but it can be done for the complete file system. The result is called a *disc cache.*

An interesting buffering technique which can be regarded as a variant of multiple-buffering is used with disc arrays to provide extremely fast data transfer. We'll describe this method in more detail in the chapter *REAL-TIME DISC SYSTEMS*; in brief, a file stored using this method is "striped" over many small discs, with one or a few sectors on each disc in the array. This file can then be read by reading a sector from each of the discs in parallel, using for each disc a separate section of a very large buffer; the effect is of reading one very large block in the time taken to read a single disc sector.

REFERENCE.

IMP32 : Acknowledgments to A.M. Lister, *Fundamentals of operating systems* (Macmillan, second edition, 1979), pages 66 and 68

QUESTIONS.

An instruction commonly used to open a Unix file is

```
internalname = open( "externalname", "mode" );
```

where "mode" is "r" for read, "w" for write, etc. The internalname is declared as an integer. How does this fit into our rather complicated discussion ?

How could compilers be told about the properties of different devices ? If they were, would that have any effect on device independence ?

We wrote : "Some programming languages automatically ensure that all files are closed when a process finishes ...". How does that fit into the process state diagram ? Will it work even if a process does not finish normally ?

Double buffering works as described for a programme which either reads records or writes records. Is it still useful for a programme which does both ?