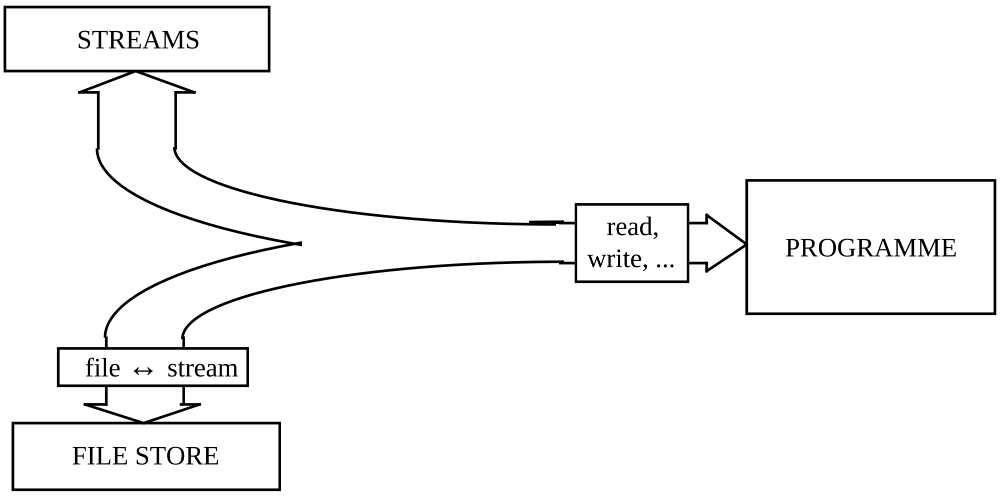
STREAMS

Streams crept into our discussion of files in the preceding chapter. We want to say a little more about them, because they turn out to be quite important in operating systems.

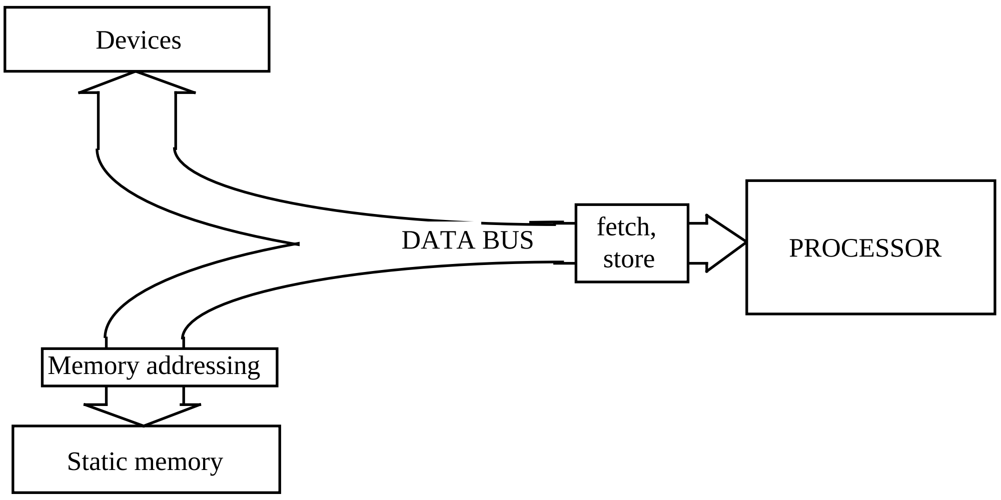
A stream is a temporal sequence of data items. We stress that it is a *temporal* sequence, because the word "sequence" is often used to denote an ordered set such as could be stored in an array and be retrieved therefrom in random order.

In fact, a large part of the operating system can be thought of as a set of stream operators. Its purpose is to organise the task of turning input into output, and it typically does so by applying reordering operators which change the order of items (anything from jobs to the execution of machine instructions) in the input stream, storage operators which save parts of a stream in a buffer for some time (think of spooling, and the various sorts of cache), and processing operators – commonly programmes – which change the material itself.

This view is the key to the relationship between files and streams. A programme is a stream operator, so it requires a stream of input and produces a stream of output; therefore, the functions of **read** and **write** instructions are to accept and generate stream elements. If the data begin or end up in files, then the file management system must handle the necessary conversions between stream and file :



It's interesting that much the same conversions happen on a smaller scale within the processor. This is illustrated clearly by the diagram below, which depicts the operation of a processor using memory-mapped devices and its relations with memory and the devices :



The items in a stream may be of any sort. Input from a keyboard is usually received as a stream of characters; in serial transmission, the characters are themselves encoded as streams of bits. Input from a (Macintosh) mouse comes as two streams – one of down and up signals from the button, and one of elementary north-south and east-west moves from the ball. At the other end of the scale, we met streams of jobs, tasks, processes, and files in the account of batch operating systems in the *BATCH SYSTEM* in Section 1.

In practice, all of these are represented by streams of data not very many bits wide, because of the organisation of computer hardware. Everything must therefore be encoded in long sequences of elementary items. That's nothing new – this text is for the most part a sequence of elementary items (characters) which represent bigger items (words, sentences, paragraphs, ...) – but all such systems require some conventions which control the way in which the larger items are represented.

Indeed, if you choose to think of it in that way, almost everything that happens in a computer system can be regarded as a stream of some sort – or, often, two streams, one of data and one of administration. When using any sort of data structure, there is a stream of data (in or out) and a stream of selection instructions or calculations to identify the position in the structure; and we could fill out both the diagrams above by adding control streams, in the first case controlling the operations of the file store and the stream devices, and in the second case typically selecting the devices and selecting the memory location to be used.

Tempting though it is, we shall not pursue this idea of great generality any further. For our purpose at the moment, we can more usefully restrict our attention to fairly ordinary streams of data – and there is still an abundance of streams once we look for them. It's interesting to observe that, though data in computer systems might spend a lot of their time just lying about in stores of one sort or another, everything that happens does so in streams. The result of a calculation is a new value, which must usually be taken away and put somewhere; to make the calculation happen, operands of some sort (from the code, if from nowhere else) had to be fetched. Nothing happens without data moving about – and a processor is just a hardware operator which merges a code stream and a data stream to produce a new data stream (and a sequence of new processor states, if you want to be picky). In many parts of computing, only the initial and final states of the system are of interest, but an operating system's job is to manage the transition between input and output, so we have to take the streams seriously.

WHERE DO THE STREAMS COME FROM ?

The obvious sources are those connected with the traditional idea of a file – that is, anything including an input or output device. Examples are devices which deal with permanent files on various media (paper tape, cards, magnetic tape, discs of many sorts, and any others you can think of) and terminals. Other ephemeral streams have become more common comparatively recently : microphones, loudspeakers, video devices, communications lines are examples.

A less obvious, but nevertheless important, source of a stream is another programme. Perhaps the best known example of this sort of stream is the Unix pipe, which is explicitly designed to channel the output stream of one programme into the input stream of another. This has been a feature of Unix from its inception, the idea being that useful operations could be constructed by temporarily connecting many simple utility programmes (called filters) to form a "pipeline". The same sort of sequential composition of operations is also useful in programmes. The idea that streams are significant for software has fairly recently become more prominent with the growth of interest in object-oriented and functional languages – for example, there is a direct correspondence between the Unix filters and C++ *manipulators*.

WHY DON'T WE TALK ABOUT THEM ?

If that's so, though, isn't it rather surprising that we hear so little about streams ? You might expect that a structure which is vitally involved in every useful programme would be a little more prominent in the average computist's consciousness. The most important reason for this coyness is seen in the final sentence of the previous paragraph : we don't think of streams if we're concentrating on initial and final states. The traditional view of computing, just a little simplified, is that we start with file A, then some magic happens, and we find that we now have file B, which is related to file A in some predefined way. This is precisely what we mean by our requirement to "produce results as instructed". In our discussion, we have started from that position, and have been led by a series of plausible, though not always explicitly stated, arguments to the notion of streams as an essential component of the system. Streams are structures which we need in order to achieve the desired end; they are at a lower level than files, and from most points of view can be regarded as details of implementation. A programmer can suppose (here's another bit of the system mental model) that a programme really operates on files, the programming language will be designed in terms of operations on files, and the operating system must support this abstraction – which it does by implementing streams.

That discussion depends on the assumption that the nature of computing is to manage a transition of the system from one static state to another. That is no more than an assumption, and it is becoming less and less tenable as time passes. Perhaps the clearest example of this development is in the area of multimedia[SUP7], where computers are expected to display continuously changing patterns of sound and vision (pictures, speech, animation, music, etc.); here, there is no stationary "final state", and the coordination of streams of data of various sorts is the essence of the "results as required". The implications of these developments for operating systems are still being worked out, but it seems not unlikely that, in at least some areas of programming, explicit stream handling will become important.

REFERENCES.

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

QUESTIONS.

To build complicated static data structures in linear computer address spaces we use devices like pointers, record lengths, record end markers, and so on. What sort of devices are available to represent complex temporal structures (characters, lines, files, jobs, etc.) in streams ?

Consider the B6700 MCP organisation shown in the diagram in the *BATCH SYSTEMS* chapter. Input to the system was a stream of cards. How could you manage the stream to identify jobs and tasks, and to make it possible to run these in any order ? (NOTE : this is the system which permitted random access to spooled card files.)

The input streams for a Macintosh system are generated by a keyboard and a mouse. How wide are these streams, in bits ? (Consider the signals they carry, and how many bits it takes to encode them.)

The output stream for a Macintosh system comes from a screen which is perhaps 500x500 pixels in size. Is the stream 250,000 bits wide ? How is this display kept supplied by internal hardware channels which are a few tens of bits wide ?

Consider our statement that "in computer systems ... everything that happens does so in streams". We believe that the statement (possibly prefixed by "almost", just to be safe) can be justified. Can you justify it ? – or discredit it ? (If it's wrong, please tell us.)
