STREAMS IN OPERATING SYSTEMS

While all the operations described in the previous chapter are initiated by the programme, they cannot be performed without much help from the operating system, so we must now determine what lies at the other side of the buffer and file information block. It is up to the operating system to make sure that data transfer and file stores are properly managed, so it must provide procedures to do so. These must be made accessible to programmes in general, typically through supervisor calls for protection.

The system must therefore :

- Define the form of the file information block, and how this is to be connected to the representation of files in programmes.
- Provide procedures to implement the permissible operations. These will normally rely on the file information block for any information they need about the file, and be accessible as supervisor calls.

CONNECTING PROGRAMMES TO SYSTEMS.

We have dealt with the programmes' transactions with the stream abstractions provided as the file information block and buffer; now we must consider what facilities required by the programmes must be provided by the operating system. Our remarks here are therefore complementary to those under corresponding headings in the previous chapter. We return to the question of how these facilities can be provided later in the *IMPLEMENTATION* section.

Opening the stream.

Information about the real stream or file corresponding to the specification given by the programme must be acquired, and combined with the programme's information to complete the link – or, of course, if the programme requests something impossible (random access to a stream, a non-existent file, a request to use a device which is already active, etc.), appropriate error signals must be produced and communicated. This will usually involve setting up some sort of link between the programme's file information block and the *device descriptor* which contains information about the physical device to be used.

It might also be necessary to perform certain actions on the device itself, or on structures used in its administration – a link might have to be established to another computer through a communications line, a tape might have to be mounted and checked, a plotter might require manual attention to load new paper. The nature and details of these operations depend on the device concerned, and it isn't sensible to build them permanently into the operating system (or you would never be able to attach a new device). Instead the requirements of different devices are taken care of by individual *device drivers*, which we shall discuss further later on. For the moment, it is sufficient that there is something which the system can use to establish the required link. (Or, more precisely, that there *must be* such provision if the system is to work sensibly.)

Depending on how the details of the system are set up, there might also be a certain number of additional housekeeping tasks to perform – read the first buffer load, mark the device in use, etc.

Reading and writing.

From the system end, these operations reduce to maintaining the buffer in an appropriate state. When an input buffer is empty, it should be filled; when an output buffer is full, it should be emptied. Again, details of these processes depend on the device and will be defined by procedures in the device descriptors, but the system must ensure that information is transmitted to and fro as required.

In simple cases, there might not be much more to the routine input and output than is implied by the previous paragraph, but not all cases are simple. If so, the complications

are generally handled through the file information block, and we discuss these further under the next heading.

Navigation and control.

The significance of the file information block can be illustrated by the common, and important, example of a disc file.

We have established the idea that a file can be seen as an array of records; clearly, the array index – the ordinal number of the record in the file – must be defined before we can read or write a record. The minimum requirement is that the record number be supplied with each request for a **read** or **write** operation. If we are willing to remember in the file information block which record we used last time, we can also provide for serial reading and writing, when the next record is automatically selected; less commonly, we can provide reverse reading and writing, where the records are used in reverse order.

The serial **read** and **write** operations convert the file into an input or output stream. They can alternatively be thought of as ways of implementing the stream operations **get** and **put** (names stolen from PL/I) on files. We do not know of any names for the corresponding operations which traverse the file backwards, though they obviously also produce streams – as, indeed, does any means whereby a sequence of records can be automatically derived from a file. Perhaps **teg** and **tup** ?

Notice, by the way, that, because of the possibility of access to records in arbitrary order, not even the simple serial copying of records from a file to a programme can be regarded as a primitive stream transaction, and the stream of requests from the programme must always be accompanied by a parallel stream of navigation information.

COMPARE :

Lane and Mooney[INT3] : Chapter 12; Silberschatz and Galvin[INT4] : Chapter 10.

QUESTIONS.

Our suggestion of **teg** and **tup** as names for stream operations which read files backwards was a joke. (It's good to get these things clear.) Apart from the silly words, though, would it be a bad idea in principle ? And, if you wanted to use something like **get** and **put** in such circumstances, what would be a better way of doing it ? (HINT : consider the roles of design and implementation.)
