SHARING FILES.

So far, we have implicitly assumed that files are private, so that there is never any ambiguity over who is doing what to the file. This is really an oversimplification, because it is quite often helpful to let two or more programmes use a file at the same time.

In the simplest case, several programmes might require to read from the same file simultaneously. This is common with files containing public software or information – help files, compilers, information retrieval systems are examples. Such files rarely change, and any changes can easily be made independently of the public access to the files, and installed at some convenient time when nothing is happening. In such cases, it is often sufficient simply to make the file read-only; the concurrent read operations don't change the contents of the file, so there is no danger of any programme's affecting any other programme's behaviour.

The next step upwards in difficulty is the file which contains current information which must be kept up to date, but which – apart from the insertion of new information – is only read. Now we must allow for one programme to write the file, while others read, and it turns out that there are several solutions. Which we choose depends on factors such as the urgency of the information (must new items be available to *all* requests immediately ?), the frequency of use (is there likely to be a gap in the stream of requests soon ?), the importance of integrity (does it matter if a process reads a garbled record from time to time ?). Generally, though, some sort of restriction on access to the file must be imposed on the reading programmes to provide a safe time for the writing programme to change the file.

The most complicated case is typical of database systems, where many programmes might be reading from and writing to a file essentially simultaneously. From the system's point of view, this is rather like the previous case, only more so, and solutions are similar but even more stringent in their requirements for access restriction. Few practical operating systems provide for such requirements (database systems usually do it for themselves), though as computer systems generally become more complicated, file systems are also evolving to cope with more demanding requirements.

If we want to design our system to cater for these requirements, what do we have to do ? A first principle is that, as the complications are standard, they should be handled by the system as far as possible, without placing needless demands on the programmer. On the other hand, the system must find out what sort of complexities are to be expected, so there must be some means for the programmer to tell the system what might happen.

The simplest way to impart this information is perhaps by an extension of the file access descriptions. The basic access mode information is still important, but we can also specify whether or not multiple access for reading and writing should be permitted, and what sort of restrictions must be enforced. With this information, the system is in a position to decide what procedures to use for the file access.

We do not propose to go into detail on how these different access methods can be implemented, but it is interesting to point out one or two general principles. First, we remark that there is one piece of information which is essential in any system of this type : the system must count the number of programmes currently using each file. Many of the file operations we discussed in the previous chapter are unwise or impossible unless the file is not in use, so the system must be able to determine when this condition is satisfied. Most simple systems have some sort of "in use" flag in their file tables; in a system which provides for shared files, this becomes a "how many users" counter.

A second point of principle concerns how different programmes handle the main data structures used in file operations – the record buffer and the file information block. Generally, as the demands on access become more restrictive, the structures have to be shared at higher levels. With multiple read-only access, no such sharing is necessary, though it might be convenient as a means of implementation. In the more complicated case where a writing programme might change significant characteristics of the file (such as its length), other programmes using the file should have access to the changes, which means that they must be able to see a common file information block. With multiple write

access, or in cases where new information must be instantly accessible, the programmes must also share the buffer.

COMPARE :

Lane and Mooney[INT3] : Chapter 12; Silberschatz and Galvin[INT4] : Section 10.3.

QUESTIONS.

What would you have to do in order to install changes to public read-only files without interrupting access to them ?

There are (at least) two ways for programmes to share file information blocks or buffers : the programmes can be given access to a shared memory area which contains the common block or the buffer, or they can retain separate blocks or buffers but have the system guarantee very fast propagation of any changes between the different programmes' versions. Which of these is better ? (NOTE : we tell you about shared memory in the chapters on memory management which follow soon, but the details aren't necessary to answer this question – just assume that it can be done.)
