HOW PROGRAMMES USE STORAGE

To continue in the proper spirit of our top-down analysis we must not take things for granted. We must analyse – and to do so we ask some questions : just what sort of storage do the programmes want ? – how should it be presented ? – and when, and for how long, is it required ? Given this information, we would hope to be able to design storage systems for our operating systems which will provide the required facilities.

WHAT WE WANT.

In the early days of computing, those questions would probably not have been understood. Everyone knew that a programme's storage was called the store (in Britain) and memory (in the USA), and that it was an **array[0 : 2^n-1] of word** (or perhaps an **array[0 : 10^n-1] of character**, or some similar variant), and there wasn't much more to say about it. There was also a rather remote permanent store where one thought in terms of files, but this was thought of as a **list[1 : end] of something** without much room at all for structure. (It was a list rather than an array because on a magnetic tape you could get at it only serially; the appearance of discs didn't change that notion a lot for some time, because though you could address sectors in arbitrary order it was regarded as inefficient.)

For our current exercise, we urge you to banish from your mind any trace of these ideas, because they are derived from knowledge about the implementation of the computer system in hardware. We shall have to deal with that in due course, but now we are hoping to find what the programmes require, not what the hardware provides, and the two are by no means necessarily the same thing[SUP12, SUP17].

How can we find out what programmes require ? One way is to inspect programming experience to find out how it is done in practice. This is a valuable source of information, but is inevitably biased by the programming systems currently in use, so we would like some less tainted source to check our conclusions. We can get some idea of programmes' requirements in the abstract from studies of programming in principle, and from attempts to design specification techniques to be used in programme design.

Our aim is to determine what sorts of storage facility we require in the operating systems, so we shall be interested in two sorts of information. First, quantitatively, we would like to know something about how much storage programmes will require, and when they will need it; second, qualitatively, we would like to know how they would like it presented. It seems fairly clear from the outset that there is no single or simple answer to either of those questions, and consequently no perfect design for our systems, but we would still like to find out what we can about storage use so that we can determine what level of help makes sense, and try to build that into our designs.

WHAT WE KNOW.

One thing we know from the start is that both programmes and data require storage, because by definition we have nowhere else to put them. The programmes themselves will not obviously cause much trouble, because they already have storage; each programme must be in a file before we can use it. The only additional task there is to make sure that the programme is in a position where it can be executed when the time comes. The data are another matter entirely, and require further thought.

The characteristics of any programme's requirements for storage depend, reasonably enough, on the programme. We know from experience that they can also depend on the data; while some programmes will always operate within the same area of memory, others will use more or less according to the demands of the moment.

We reach the same conclusion by considering what we know about programming and programme design. A very early study of how to write down algorithms was published before there was anything much like an operating system; that was the Algol Report[SUP13]. While it was written with programming in mind, Algol was originally design as an ALGOrithmic Language, and was only later thought of as a programming language in its own right. By considering the sorts of instruction we wanted to give when describing an algorithm, the authors worked out a set of basic instruction forms, which evolved into what we would now think of as a programming style. This was enormously influential, and has been used, more or less unchanged in principle, ever since; Java fits in very well. This presumably means either that they got it right, or that no one has come up with anything better, and either way it should be a good guide. The main principle which concerns us is that programmes are not simply featureless collections of instructions, but come in modules of some sort, and these modules are executed with some measure of independence.

Essentially the same conclusion turns up everywhere. For example, the language ZSUP14, used for formal specification of systems – not only programmes, but in principle almost anything – relies strongly on the same ideas of modular decomposition, and in the case of programme design these are essentially the modules introduced in the Algol report. Whether this is something inherent in large systems as a matter of principle, or whether it is merely a reflection of the way we address large systems, it seems to work, and is a reasonable starting point for our current discussion.

The different approaches also agree that, in execution, modules are not necessarily used according to any simple pattern. Some persist for a long time, some are used only fleetingly. Modules call each other (or even themselves), and the order in which the modules are executed can be completely determined by the data given to the programme. The amount of storage required by a module can also depend on the circumstances – and it might or might not be released when the execution of the module ends.

The more theoretical treatments typically offer little in the way of suggestions on what happens within the modules, but experience with programming shows that in practice we use a great variety of data structures, typically composed by associating simpler structures using some form of link. In this way, large and complex structures can be formed, in which there might be as much valuable information in the pattern of the links as in the data which are linked together. Further inspection shows that, as we suggested, programmers have had to find ways round the system-supplied memory management to implement the structures they required. In Fortran, you declared a very long one-dimensional array and built the structures within it using array index values as links; Pascal gives some help by providing a *heap*, within which structures can be built according to your declarations using the **new( )** function. Languages such as Algol68 and PL/I provided heap-like facilities, and specialist languages such as Lisp and Snobol implemented very flexible memory management schemes right from the start. In almost all cases, though, such systems were implemented within the language, with practically no help from the system's memory management facilities.

Can we conclude anything useful from this account, other than that almost anything can happen, and probably will ? We think that we can. Our discussions have given us a picture of a programme in execution requiring temporary storage areas from time to time, with sizes determined by the nature of the programme and data. We shall call these areas *segments*. Once provided, the programme will have to be able to find the segments, so some sort of identifying mechanism will be required. Here, the programme's position is much the same as that of someone needing an identification mechanism to keep track of a set of files, and similar arguments apply, with the difference that only a temporary association is necessary and the "names" need only be intelligible to the programme. We might therefore expect that the programme will be associated with some sort of table of its segments, with some sort of names or addresses as contents.

Noticing once again that these observations are of widespread, if not universal, application, we see that running programmes require services which segments of storage from time to time, and that these requirements are not trivial. The universal need suggests that the appropriate agent to provide these services is the operating system, where they can be implemented once and for all and made available to all programmes.

Finally, it is worth pointing out that nothing we have discussed suggests that there's any significant distinction between storage used by files and storage used by programmes. The closest to any such suggestion was the observation that the storage used for the programme must be accessible to the processor, but the comments on structures apply just as well to data in files as to data in use by a programme – after all, the data in the files must have been used by a programme and is going to be so used again.

REFERENCES.

SUP12 : A. Kay, in a discussion : *Sigplan Notices* **32#9**, 15-37 (September, 1997), page 16.

SUP13 : P. Naur : "Report on the algorithmic language Algol 60", *Comm.ACM* **3**, 299 (1960)

SUP14 : B. Potter, J. Sinclair, D. Till : *An introduction to formal specification and Z* (Prentice Hall, 1991).

SUP17 : H.W. Lawson : "Salvation from system complexity", *IEEE Computer* **31#2**, 120-119 (sic) (February, 1998).