CONTROLLING THE COMPUTER SYSTEM

The operating system is a device for obeying instructions. It controls a variety of hardware and software resources which it can dispose as required when it receives instructions, and we've seen that the instructions can come from several different sources : from people, presented through the terminal; from files containing instructions; and from programmes as they run. Here we enlarge on these sources, and how the operating system can deal with them.

CONTROL FROM TERMINALS.

Instructions coming from the terminal are perhaps the simplest sort. They usually involve one, or at most a few, programmes and steps of operation; and if anything goes wrong the system can simply report the problem back to the terminal and rely on the person running the job to sort it out.

Translating the input signals from keyboard, mouse, light pen, dataglove, or whatever into instructions to be executed is the job of the *user interface manager*; we shall discuss this component of the system in some detail later.

CONTROL FROM COMMAND FILES.

As well as such direct "hands on" control of the system, we also want to be able to collect sequences of instructions and save them for execution later. This might be either because we have found sequences of tasks which we often do, or because there are tasks which must be carried out automatically or at inconvenient times. In either case, we want to write a job-control programme which can be executed by the system as if we were entering the instructions at a terminal. Such a programme might be called a *job-control macro, shell script*, or *command file*. We shall call them "command files", because it seems to be the most common name. It is perhaps self-evident on grounds of the desirability of consistency (though in practice not always implemented) that the language used to write the programme should be as close as possible to the ordinary job-control language used in interactive work.

That is a fairly straightforward thing to arrange in a conventional system with a traditional job-control language. It is far from clear how to provide the same facilities in a system in which instructions are issued through a predominantly graphic interface. Even if we restrict ourselves to text, there are many questions. Three main design requirements give rise to a good proportion of these. They are the need for a flexible system, the need for safety in case of error, and the need to cater for both the interfaces (programme and operating system) associated with the input stream. The next three paragraphs present a selection of the implications of these requirements.

We need **flexible** systems to cope with changing circumstances and changing environments. A command file should be able to accept *parameters*, so that it can at least be used with (say) different files at different times. The job-control language also needs *conditional constructs* of some sort, so that the command file can be written to test for and cope with different circumstances – such as missing files – in its environment. **if** and **goto** will do, but proper loops and subroutines are better. As such structures are not obviously needed in ordinary interactive work, the job-control language must be extended.

To write **safe** command files, we must be able to find out what's going on in the system. We require good access to appropriate system tables – particularly *file system information*. We also need good *communication with the programmes* we execute, so that we can pass information to a programme, and also find out whether the programme succeeded, or, if it failed, why.

Catering for **both** (command file and terminal) **interfaces** can be considerably more demanding, depending on details of the way the system handles its terminals. Generally, though, only in the simplest systems is it a matter of merely replacing input from a terminal by input from a file : in fact, that apparently simple implementation technique might be impossible if there is no system-wide terminal handling software. We need means of *directing instructions* to either programme or system; and we need *access to the programme's output*, so that we can take action if things are going wrong.

We shall say more about these questions when we discuss implementation issues (in the chapter *TERMINAL LOGS AND COMMAND FILES*); for the moment, the conclusion is that a good command file system is rather hard to get going – and probably impossible without a repertoire of textual instructions. We shall say more about the job-control languages themselves in the next chapter.

CONTROL FROM PROGRAMMES.

Whether or not system facilities should be accessible from programmes which aren't part of the system has been a question of degree rather than kind at least since monitor systems began to change into operating systems. Before the change, normal programmes were seen as quite separate from the system; although subroutine libraries were often provided to handle (particularly) input and output for people who didn't want to write their own routines, these were independent of the system's own input and output routines (despite the fact that they could well be essentially identical in their code !). With operating systems, such freedom was no longer permitted. Instead, people were constrained to perform all their input, output, or other sensitive operations through system procedures of some sort, and protective measures (supervisor calls, etc.) were developed to protect

the system against unwarranted intrusion. Nevertheless, access is still possible, if only to selected parts of the system. Is there any reason why we should prohibit such use ?

The ideal answer is perhaps "no, unless it could be dangerous for the system in some way". In early operating systems, the facilities were often not available – either the designers had decided that they would indeed be dangerous, or (more probably) hadn't even noticed that there was a question, assuming that only operating systems needed to use any special facilities, and that the system programmers could get whatever they wanted anyway. Later systems took a more relaxed view, and provide means of acquiring information about the state of the hardware, the operating system, the file system, and so on, and of requesting various system services from programmes, ranging from changing the protection codes of files to running other programmes. More recently, it has been argued that developers should be *encouraged* to use the system's procedures in the interests of compatibility. This leads on to the idea of the application programmer interface, which we mentioned earlier; we shall discuss this further in the chapter *DEFINING A SYSTEM INTERFACE*.

The facilities have usually been provided as a set of subroutine calls, commonly of daunting complexity. While there is some excuse for the complexity if the service required is intrinsically complex, and is not usually visible (perhaps changing memory management procedures, for example), it is less defensible if all we want to do is something for which there is a simple system instruction – say, removing a set of files. In commenting on scripts above, we suggested that it was self-evident that they should be expressed in the same language as that which one would use from the keyboard. Why doesn't the same principle apply here ? Why can't we just write something like `DO( "rm ss*" )` ? After all, all the machinery is there somewhere. In a few systems, we can do just that (in Unix, we can indeed now write `system( "rm ss*" )`, but it's a fairly recent addition); but in others, we must either go through a long rigmarole of calling a file system procedure to expand `ss*`, then another, repeatedly, to remove the files – or we must go through a different rigmarole to execute the `rm` programme with parameter string `ss*`.

It's interesting that something quite like our suggestion was provided by the early generation of 8-bit microcomputers, though by no means all provided anything recognisable as a command file. In a common pattern of use, the terminal would accept either Basic language instructions or system instructions, and system instructions could appear in Basic programmes. The view among serious computists at the time was that this was a hopelessly amateurish approach, but what could you expect from toys ? In the *INTRODUCTION*, we asked whether such systems might have been taken more seriously if our "service model" of operating systems had been adopted; we don't know the answer, but it does seem likely that some lessons which could have been learnt from these early systems had to be laboriously rediscovered later.

The first close approach to our `DO( )` instruction which came to our notice was implemented in the BBC microcomputer's Basic. It was called `OSCLI` (Operating System Command Line Interface), and accepted a string variable, which could simply be an instruction between quotation marks, or it could be tailor-made using the Basic string operators. It worked.

QUESTIONS.

What would have to be done to implement the DO( ) function ?
