STARTING A PROGRAMME

The first requirement for running a programme is to get it into memory and point the processor at it. Accordingly, we're going to begin with a fairly brief description of some of the practical implications of this rather obvious preliminary. It will be brief, because, though we can describe *what* has to be done, we are not yet ready to go into full details of *how* to do it. We think it's worth presenting the description to set the scene for what follows, but you should remember that we're describing the bare necessities; the operations described are about enough for, say, a rather simple monitor system, where only one programme is concerned and there are no problems from memory managers or the need for complicated device management. Even so, the description is useful, for these actions have to be performed somewhere even in a more elaborate system.

The story starts when the system receives an instruction to run a programme. This might come from a terminal, either as a text string or as a sequence of mouse operations, or it might come from another programme; whatever way, it must identify the source of the programme to be run (almost invariably a disc file), and it might include parameters which are to be presented to the programme when it starts.

Having received the instruction, the system must read the code from the disc into memory. The parameters, if any, must be communicated to the programme. Finally, the programme's entry point must be determined, and the processor is then caused to branch to the entry point, whereupon execution begins.

None of these steps is trivial, though the first – reading the code into memory – is covered by parts of the system we have already discussed. For the others, some system conventions must be established, and either promulgated as rules to be followed by developers or built into the API as procedures : means of communicating parameters, and how to record the entry point in the code file, are examples. Notice, incidentally, that this is one case in which the system must know something about the contents of a file.

That is approximately all that is needed. "Approximately", because different systems have different requirements, and these might require attention during the programme loading operation; "all", because that really was (approximately) all for a monitor system.

What does that tell us about the process table ? Not a lot, but enough to get started. Two points which we established in our earlier discussions are important : that the process table is composed of one entry for each process; and that this entry is the root from which we should be able to find out anything we want to know about the process.

The first step is therefore to find a place to build our new process table entry. Immediately we have a question : should we build a compact table, with all the entries (which we shall call *process control blocks*, or PCBs) collected together in an array, or would it be better to keep the process control blocks in or near the memory areas used by the processes, and link them by some form of pointer ? The quick answer is that it doesn't really matter much. Both methods have been used, and work, but our impression is that compact tables are more common, so that's what we'll assume. With that structure, the system always knows where the table is, which makes it very easy to search the table if we're looking for something; on the other hand, to start our new process we have to find a vacant space (so vacancies must somehow be indentifiable), and there might not be one (because the table, if fixed, can't be of arbitrary size). Observe how practicalities crowd upon us as soon as we venture out of the realm of pure design !

Assuming that we find a vacant place in the table, what shall we put into it ? One thing we shall certainly need to find is the process's code, so one entry will be the address of the memory area - or, more likely in a larger system, the address of its addressing table (which we shall of course also have to set up). A pointer to a parameter string will also be useful if parameters are used. We shall also want somewhere to keep the current value of the programme counter while the process is not active - which is how all processes begin, because we can't start them until we've completed the administrative jobs. Finally -

for the moment, for we'll find many other items which can usefully be kept in the process table - there is a rather less obvious item; some sort of process identifier. This has no specific function in the setting-up activities we are discussing, but it is important to be able to identify a process in various sorts of message and in other sorts of link in the system. Early systems used the index of the control block in the process table, but this turns out to be unsatisfactory if trying to trace the history of the computer's operations over a length of time; a unique identifier which will not perhaps be allocated again to another process immediately after the current process dies is quite important, so typically a number of four or five decimal digits is used.

That gives us considerable help in organising the starting-up procedure, for we can now rephrase the sequence in terms something like this :

1 : Acquire a vacant process table slot in which to construct the new PCB.
2 : Allocate a process identifier, and store it in the PCB.
3 : Get some memory, and put its address in the PCB. (Or do the rather more complicated, but equivalent, thing with an addressing table.)
4 : Read the code file into the memory.
5 : Find the initial programme counter value, and store it in the PCB.
6 : Put a pointer to the parameter string, if any, in the PCB.

(We emphasise "something like this". We'll see later in *PROCESSES IN ACTION* that there can be considerable differences between different systems' ways of building processes, but the aim is always a properly constructed PCB.)

Now we have a process set up in a perfectly standard form which is recognisable to the rest of the system. To make it run, something (which we shall eventually call a dispatcher) has to load the programme counter from the value in the PCB, and there we are. Easy.

Are we cheating by leaving out the hard bits ? We've certainly left out some bits, but that's only because we haven't yet discussed all that we need. It really is (quite) easy, because of the structure imposed by the process table.

QUESTIONS.

Consider the sequence of actions required to start a programme. For each step, ask :

- What is it that tells the system it has to take action ?
- What information does it need to carry out the action ?
- Where does the information come from ?

We've assumed that the file to be executed is a machine code programme. Suppose instead it's something which has to be interpreted (such as a command file). How does the sequence above change ? (NOTE : It's still easy, but just a bit different.) What is the process ? How is that related to our ideas about execution put forward in *PROGRAMMES* ?
