PROCESSES IN COMPUTERS

Our tentative ideas of processes have helped us to organise the machinery for starting a programme, but before going any further we should pause to make sure that we know what we're talking about; it is not self-evident that ideas gleaned from an amateurish model of a bakery can be carried over unchanged to the operation of a computer.

In fact, though, we did not choose our examples at random, so they have reasonable analogues in the realm of operating systems. Here are some examples which you might like to compare with the bakery examples in *PROCESSES*; they're not intended to be identical, so look for baking equivalents of the computer examples, and vice versa.

- First, two people might execute the same programme at the same time. (We don't mean a programme which is designed to be used by several people cooperatively – that would fit better into the third category. Think of something like a compiler which many people want to use.) There is no connection between the two people's work, and we certainly want to speak of them as distinct activities. Perhaps it would be useful to have separate memory links for code and data in the process table so that we could use the same code without making a copy for each process.

- Second, a single task might involve more than one programme. Indeed, we shall see that that's essentially what happens every time you start a new programme in a Unix system. Do we use the same PCB, or start a new one ? It's a continuation of the same activity, so there's no reason why we can't keep the same one. In older systems, it wasn't uncommon to find a programme instruction often called **chain** or something of the sort which you could use to switch smoothly from one programme to another as a part of the same activity. (Whether or not this is possible in a particular case depends on the designer of the operating system. There is no reason why new programmes must be started as new processes. It is often necessary for the operating system to change the resources associated with a process as it executes; the code used by the process is just another of these resources.)

- Third, as multiprocessor computers become more common, there is increasing interest in speeding up execution by exploiting parallel processing *within* programmes – so your programme might acquire half a dozen more processors and split itself into seven strands for a while. Each processor is industriously executing its code stream, but it is only doing one seventh of what the original processor was doing before the split. (Notice that, from the point of view of the programme, nothing new in principle is happening. Unless the separate strands interact, the same (or almost the same) code could be run in series on one processor. Seen in this way, all parallel strands of execution within a programme are incidental parts of a single task.) Does that require more PCBs ? The static structure of the code and data remain unchanged, and so far as the bits that live in the PCB are concerned the main difference is that you need several different programme counters, one for each strand. (We haven't mentioned dynamic structures like the execution stack yet; they will have to be multiplied too.) In practice, that's probably enough excuse to make new PCBs, but we might expect that they have to be connected together in some way to show their relationship, so we must provide a linking field in the process table entries.

- Fourth, two separate programmes – which may even be running on different computers – might cooperate to complete some sort of job. (Think of a network file server. The task to be accomplished cannot be completed without the contributions of both programmes, even though they are separate entities in the system.) Should we try to describe this interaction by a single PCB, or would it be better to think of separate activities which can communicate ? In practice, it is usual to choose to keep the processes separate, but the alternative is there.

These are samples from a wide range of possible requirements. It is important to consider such possibilities, because however we choose to represent processes our method must work in all cases - or, to put it another way, a poor choice of process representation can

make it difficult to implement some useful sorts of process organisation. We have sketched possible responses to the questions raised by each of the examples, and we have been able to do so without any major change in the process table idea; in practice the process table model does seem to be a very effective way to keep track of what's needed, and we shall assume it from here on.

So we had better decide what we really mean by a process. We have got so far on the merits of analogies with cake baking and general knowledge about what goes on in computers, but we have certainly been making up answers as we go along. We have found a few cases where we can choose whether or not to make a new process control block, and, by implication, a new process, but we've had no guidelines as to which might be preferable. We have a vague notion that a process is a basic unit of computation, but we don't know how to work out how many processes we want to execute a particular piece of code, nor how they fit together. Can we do better ?

PROCESSES AND PROCESSES.

When we come to analyse these structures more carefully, we find ourselves once again in an area which looks rather different from different viewpoints. We discovered that for a full discussion of memory management, we had to see the problem both from the point of view of a process (whatever that is) and from that of the system; we made a similar discovery about files, which led us to invent streams, and we'll do it again for the disc system when we come to discuss its implementation in more detail. Now we're about to examine programme execution in a similar way, and once more we'll find it useful to speak of two views.

Why does this keep happening ? It's because of our attempt at top-down analysis, and our close approach to the real computer hardware. At each point in the discussion, we are considering some facility which we have decided that the operating system must provide (in this case, certain patterns of programme execution), and wondering what lower-level machinery (hard or soft) we need to make it happen. At the higher levels, we can often just invent a plausible lower-level mechanism without much constraint, but now that the lower level is very much constrained by the hardware available we have to make sure that any proposal we make really can be implemented – which is to say, we have to evaluate it from the hardware viewpoint.

The views in this case are, first, what we – the programmers – want in order to construct useful programmes effectively and easily, and, second, what the hardware provides.

The four descriptions above are examples of things that programmers might like to do. They all – unsurprisingly – fit in with our fundamental aim of getting work done, and they all share the idea that a good way to get work done is to make a processor execute a sequence of instructions. But "work" itself is not a continuum; we see it as a collection of jobs to be carried out to completion, with each job having some internal cohesion which distinguishes it from other jobs. This idea of completeness is important to us in thinking about our work, and it will help us if we can carry it over into our computing activities.

We also know (structured programming again) that completeness is a hierarchic quality. We have found that a good way to tackle a complex task is to split it into simple ones, and to keep doing so until we come to a level at which the structure of the task is so simple that we can easily decide how to write the code which will do what we want. Certainly, from the design viewpoint, the acts of executing these fragments of "doing work as instructed" are in some sense the atoms of our computation, and we shall call them *elementary tasks*. To make all this work, then, we need some means of executing

the fragments of code which we write at the bottom level of the analysis, and of tying them together in the right order. Sometimes the order of these elementary tasks is important; sometimes, as in the parallel processing example, it might not be, but provided that we can get the right order when it is wanted, that's all right.

The elementary tasks are associated with pieces of the programme. That is not to say that the collection of tasks is the same as the programme. In executing a programme, it is unusual for all its tasks to become active. Instead, we can think of the programme as a collection of potential tasks, some of which are selected during each execution according to the vagaries of the programme and the data given at the time.

Are these the tasks we wanted ? Almost. The elementary tasks we've described are certainly the raw material from which we can construct the rather larger tasks described in the examples. The main difference is really a matter of usage. We have used terms in such a way that each of the larger tasks must be constructed from one or more elementary tasks. In the first example, each execution of the programme is a different task, and the two are quite likely to include different selections of elementary tasks. In the second, the tasks happen to include examples from two different programmes. In the third, it turns out that some of the elementary tasks can run concurrently without interference. In the fourth, the elementary tasks again come from different programmes, but this time the programmes must run concurrently. Clearly, in defining the elementary tasks we have identified something closely related to what we want, but there is nothing in the descriptions of the examples to identify it more precisely.

Perhaps the view from the processor will be illuminating. What is it like ? Rather simple. At the lowest possible level, it is quite boring. Almost all the time (provided that it is running at all), the processor is executing an instruction, which tells it, implicitly or explicitly, where to get the next instruction. And that's it – except for interrupts. And (if you're a processor) the main excitement in an interrupt is that it is another way to determine where to find the next instruction. You don't know that an interrupt will mark a significant change in activity, which is likely to imply a new (*still* undefined) process. There's certainly no reason to suppose that an interrupt is an essential part of switching from one process to another, so – from the processor's viewpoint – the change could be quite imperceptible.

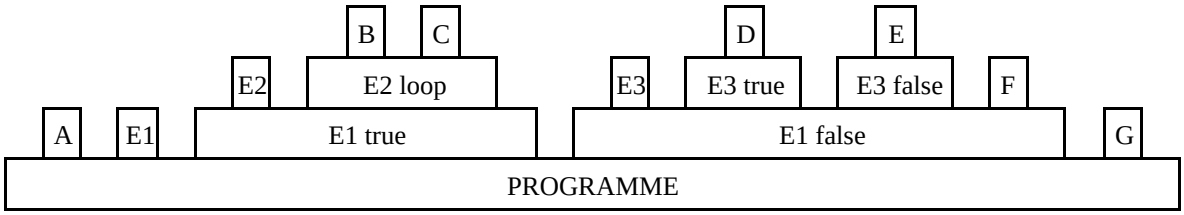
The significance of this observation is that if we want to define processes then we must do so at a level higher than the basic hardware. In fact, we have already argued that we *do* want to define processes, and we are trying to do so. In this search, there is more to be gained from our expectations of how the processor will operate, and, in particular, from our requirement for multiprogramming. It is not an accident that three of the four examples are of situations in which two or more things are going on at the same time, for it is unlikely to make much sense to try to discern processes where only one thing is happening. If we are trying to execute several processes, though, we expect that they will from time to time be stopped (for example, to await service from the virtual memory system) and then carry on from where they were halted. The *system*'s action in directing the processor's attention from one process to another is sometimes called a *context switch.*

It is here that the notion of a process becomes very important, for it is the process which carries the sense of continuity in the execution of any activity. But at what level do we wish to apply the label "process" ? The notion of continuity and completeness is there at all levels, as much at the job or programme level as at the level of elementary tasks, which correspond to the segments of the code. In the abstract, it might be appropriate to regard all levels as processes and to recognise that processes are nested in much the same way as procedures are nested.

As a guide to implementation, this is not such an attractive idea, because it requires that we should maintain many entries in the process table for each computation, which would require links to show their relationships. On closer inspection, though, it turns out that much of this obvious structure does nothing useful for us. Consider this (somewhat contrived, but adequate) example of a programme :

```
A;
A;
if E1
then begin
     while E2
     do   begin
          B;
          C;
          end;
     end
else begin
     if E3
     then D
     else E;
     F;
     end;
G;
```

Each piece of code represented in capital letters is one of the elementary tasks which we mentioned, and we suppose that the complete task is called PROGRAMME. As this code is executed, the corresponding process stack will expand and contract as shown in the diagram below, where the stack is shown growing upwards, and time increases (with potential repetitions around the E2 loop) from left to right.



Up to four process control blocks will be used, and there will be a significant expenditure of effort in maintaining the process table. But what will be in the four blocks ? Consider the situation when process B is active. E2 loop, the E1 true block, and Programme are all simultaneously active, and it is certainly reasonable to think of each of these as legitimate subtasks within the programme. But much of this information is redundant - for if B is active the structure of the programme guarantees that the processes below cannot but be active, and the information we might keep in the PCBs for the lower processes adds nothing to that held in B's PCB.

It is therefore legitimate to limit the complexity by maintaining only one PCB for a case like this, which is usually regarded as PROGRAMME's control block. If we wish to articulate the principle behind this decision, we can say that *we identify the process as the highest level elementary task which we can choose without losing any significant information.* If we choose the higher level, we have no representation in the system of the finer details of task structure, but provided that has no ill effects it might be a sensible choice. In practice, we find that the higher level, corresponding to the programme, is a helpful abstraction, and this is what we usually choose as the model for a process. This brings us back again to the idea of work as composed of distinct jobs.

The process is therefore that sequence of operations which corresponds to the execution of a programme, and which, even though prematurely stopped, must be able to start again as though nothing had happened. The process is the abstraction of continuous execution which we (the programmers) wish to be able to assume so that we don't need to worry about the elementary tasks being interrupted, and the operating system's job is to implement this abstraction so that the programmers can use it, no matter what the processor happens (or processors happen) to be doing at any moment.

The identification of process as a processor executing a programme works well in many circumstances, but sometimes it falls short. This is notably true in the case we mentioned earlier, where a single task splits into several parallel tasks. Here, we are concerned with structure within a process which is significant in its execution; we are coming closer to the level of the elementary tasks we identified earlier. It is reassuring that

our definition above still serves well; to definite the process at the level of the programme would clearly lose important information, so we must identify the separate lower-level tasks as the processes. As we saw, these are also characterised by continuity and completeness, but we regard these potentially parallel operations as more mutually dependent than separate programmes. (Even here, there is a continuum : consider the cooperating programmes in our fourth example.)

Should these tasks also be called processes ? If we base our definition on the "abstraction of continuous execution" criterion, then they certainly are – but they are not as aggressively independent as processes which are executing different unconnected tasks. They are sometimes called *lightweight processes* or *threads*. Here we shall use the word "process" to mean a heavyweight process, and speak of "threads" when we wish to refer to the lightweight processes. We shall also suppose that, as is still the common case, a process has only one thread unless we explicitly state otherwise.

The reason for making the "heavyweight" and "lightweight" distinction is the significant difference between the administrative overhead for setting up a process from scratch and for setting up a new activity within an existing process. Details will follow, but in general to set up a process the system must find a code file, allocate memory, and make adjustments in and links to many system tables, and this work is costly in time. In contrast, a thread within a process usually works within the resources already allocated to the process, and therefore causes very little disruption. Threads are important because if they are not much easier to set up than "heavyweight processes" then much of the advantage of parallel processing might be lost

Are two levels sufficient ? Should there be – so to speak – more levels of threadedness ? There is certainly no obvious reason why a thread should not itself divide further into more subthreads, and so on. The answers are probably "yes" to the first question, and "no" to the second; the argument is similar to that we used earlier to justify limiting the number of levels at which we represented the elementary tasks. Unless we can show that we need an additional level to keep useful information, there is no point in having it.

A second, and at least equally important, distinction is found when processing switches between threads. Because of these differences in environment, switching between two threads belonging to the same process is likely to involve very little overhead, as most attributes remain the same. Switching between threads belonging to different processes is likely to require much more extensive changes to the processing environment, as most attributes of the environment – code, data, protection status, and so on – must be changed.

The significant distinction is therefore between threads which cause sufficiently little disturbance for it to be ignored, and those which cause sufficient disturbance to be a nuisance. While the threshold might perhaps be difficult to define in quantitative terms, there is only one of them, so only two sorts of thread.

It is interesting to notice that these ideas are not new, though as we have come to understand more clearly what is involved in concurrent execution they have become clearer and more standardised. Something very similar to the idea of threads was possible in IBM's venerable OS/360 operating system, where a job could be given a large address space and many tasks could be run within the job. In the Burroughs MCP, starting a new thread was as easy as calling a procedure; with a few not very worrying restrictions, you could enter a procedure as a subroutine, as a coroutine, or as a new thread. Because of the organisation of the Burroughs' stack architecture (the stacks could branch, which made sharing address spaces trivially easy), it made very little difference to the system.

In practice, different definitions of a process are used in different contexts, and particularly in descriptions of different operating systems. The important notions are the continuity and completeness characteristic of processes, and the continuity, more local completeness, and complementarity of the threads. These are important ideas, and are in practice worth implementing.

TWO PARTS TO A PROCESS.

With the advent of affordable multiprocessors, running different threads of a process on different processors is becoming increasingly important. This highlights the two different sides of a process's nature. To represent a process the operating system must keep track of the current point of execution of each thread in the code which it is running. The operating system must also keep track of the resources to which the process has access. Corresponding to each of these tasks there must be a data structure to preserve the values when necessary.

In many systems, these two sorts of structure are combined in the process control block. That is reasonable if there is only one thread of execution and one collection of resources for each process. When a switch occurs from running one process to another there has to be a complete change of the resources known to the processor (think of memory in particular) as well as a change in the code being executed. Once we permit a process to split into several threads, though, it might make sense to have most of the same resources being shared by the threads. If this is the case, moving from one such thread to another requires a lot less work than the full switch mentioned above. Should we, then, separate the resource management functions from the execution control ?

No, we shouldn't, unless we have some more clear evidence that it would be useful. So far as we can see, while the points raised above are valid, they are outweighed by other factors. The main consideration is that once we have split a process into threads, we can't tell what the threads are going to do. It is quite possible that they will move on to execute quite different parts of the programme, when the resources they use will be significantly different. In the interests of generality, therefore, it is better to ensure that our representation for threads can always cope with anything that might be required of it, and the only way to be sure of that is to give each thread a full PCB.

The requirement that there should be just one thread of execution and one collection of resources is not quite met in Unix, but the system is so organised that each process can assume that the assertion is true – the system maintains it as an abstraction. Processes can share the same code as a matter of course, but, as the code doesn't change at all, the abstraction is maintained. We'll see a special case later (PROCESSES IN ACTION) : a newly constructed process shares the code of the process which constructed it. The processes also share open streams, which is rather harder to fix, but Unix provides special structures to make sure that the abstraction still works.

Some people treat lightweight processes as distinct from threads. They would say that a "thread" places the emphasis on the sharing of resources whereas a "lightweight process" places the emphasis on the ease of switching from one process to another. If you followed our argument above, you will see that there is really very little difference between these two notions, as it is the degree of sharing which determines the ease of switching.

When it comes to using a multiprocessor we can run multiple threads of a process truly simultaneously. We will look at this in more detail in a later section.

GOING TO EXTREMES.

Even with the threads, we have not necessarily reached the level of our elementary tasks. We have stopped at a rather arbitrary level defined, in practice, by our ideas of what might be useful programme fragments to execute (at least potentially) in parallel. Wouldn't it be better to use the elementary tasks as the units of execution, in much the same way as we use the segments as our units of memory management ? Accepting that a process's

memory demands are really determined by the segments rather than by the whole process we were able to design memory managers responsive to the real system behaviour. Perhaps if we similarly seek ways of managing processing in terms of the processing of elementary tasks, we might be able to produce process managers which would automatically execute the elementary tasks for us. With a system of that sort, we could just let our programmes run, and everything else would fall into place automatically; tasks which could run in parallel would do so, and those which had to be serial would be run accordingly.

Unfortunately, things are not as simple as that. The analogy fails because any set of segments can coexist in memory without affecting each other, but the execution of one elementary task might be strongly dependent on the execution of certain others, and we could not guarantee our functional system if the order were not maintained. We recognise this traditionally in the constraints we place on execution order by executing our programmes sequentially. In practice, the conventional programming languages overdo the constraint by forcing us to impose some order on operations which we know do not interfere with each other and could therefore be run in parallel. Parallel programming languages and techniques have been developed in order to overcome these constraints where possible, but the decision on what can be made parallel is still mainly dependent on people's decisions. (Compare memory management by overlays.) The system could only guarantee to execute the elementary tasks in the correct order if it was given sufficient information to define the order, and so far it has proved very difficult to generate this information automatically for any but the simplest cases.

The problem can be solved by changing to a rather different sort of computation. The *dataflow* computer architecture[EXE1] is designed so that each instruction can only be executed when its data are ready, and the information contained in a dataflow programme is sufficient to guarantee the correct order. This is parallel execution right down at the level of machine instructions. It seems to work (we have no direct experience of it), but is at present confined to experimental implementation.

PROCESSES AND RESOURCES.

In this section, we explore how to make this real work happen as we want it to happen. We have already said quite a lot about that in earlier sections. Some examples : we might want to use many processors; we might want to run many activities "at once" in multiprogramming; sometimes activities have to wait for certain resources to become available before they can continue. These are just some of the complications which we want to support with our simple model of a running programme. How can we do it ?

We already know how to do it; we shall use the notion of a process to guide our management of the computer's activity, much as we use the notion of a file to guide our management of its storage. We discussed processes in the abstract at the end of the previous section, and defined (in *IMPLICATIONS OF PROCESSES*) something of what that would mean. Now we must find ways of putting this abstract specification into practice. We have covered quite a good proportion of the requirements in our discussion of memory, but there remains the final item of the list : "a structure which we have called the process table will be necessary in order to keep track of the active processes and their current activities".

What we have not yet done is get down to specifics. That's because we have to consider the requirements of a real system to determine just what we want to keep in our process table. In practice, this shows up as differences between the details of process tables implemented in different systems; while there is a core set of items which appear in pretty well all the tables, there are others which some system designers find useful while others don't. We shall take the approach of following the operations of an imaginary, though plausible, system, and as we go we shall identify items which should be recorded in the process table, as well as their connections with other parts of the system.

A process is a rather abstract entity, which is one of the reasons for our difficulty in defining it, but it has much less abstract connections. If that were not so, process

management would be simple; as it is, we have to find the resources which a process needs before it can proceed with whatever it's supposed to be doing. In practice, process management is to a large extent a matter of allocating resources to processes as they are needed.

What do we mean by a resource ? It's anything which a process must "own" in order to proceed. We've quoted "own" because it means rather different things in different cases; it might imply that the resource has been allocated to the process (memory), or that the process has permission to use the resource (a file), or that the process has staked a claim which persists until the process gives it up (a lock), or that the process has received a consumable resource (a message). In all cases, though, the process has whatever is needed to continue with its work.

The basic resources are code and data, and a processor. Other resources are needed to accommodate the code and data – memory and files – and to get them into and out of the system – devices and communications channels. There are also more abstract resources, which amount to permissions, often used to manage shared resources – so a process might require protected access to a shared area of memory so that it can carry out some operation involving the contents of the memory with a guarantee that the contents cannot be changed by another process during the operation. (We shall study this requirement for *mutual exclusion* in some detail in the chapter *PROBLEMS OF CONCURRENT PROCESSING*.) Parallel threads of a single process might have to wait for all the other threads to complete before they can recombine to form a single thread again. In all these cases, something has to wait. These abstractions are implemented by locks and other synchronising resources of various sorts.

Much of process management is therefore concerned with keeping track of the resources owned by a process, finding new ones when required, and dealing with the process itself according to the resources available. You will read much more about these topics in the next few chapters.

In a system designed to run a single process, many of these concerns are less pressing, because the competitive aspects of resource allocation are no longer of the same concern. Resource management can then be reduced to its simplest : if the resource is there, use it. Even in this case, it is good policy to take process management seriously, because the techniques used provide a very orderly way of keeping track of what's going on.

> *An alternative view might be that the last paragraph should have begun "In a system designed to let a single programme run itself ...". Few small systems are designed with any very clear model of processes, let alone threads, simply because there is very little need for process management. We would argue that it is nevertheless useful, if only as an aid to clear thinking, to identify such entities. Typically, as well as the programme to be executed (rarely more than a single process) there are identifiable processes which drive devices and deal with interrupts. They have their own code and data, and communicate with each other in various ways. To recognise this structure in the system, and to take account of it when designing and constructing the system, is likely to lead to a product which is comparatively easy to adapt to new requirements when necessary; the alternative of building something which will do the job but without clear structure, though it has a long history, is less likely to give a system which is flexible.*

In designing the Macintosh system, the intention was to provide an environment in which programmes could be executed, and also in which many useful (or not so useful) facilities would be available all the time. These were (and, at the time of writing (1995), still are) called desk accessories. Like anything else in a computer, they were implemented by programmes – but, as the system was designed for a single process, they had to be special in some way. They were implemented as a rather special sort of

device driver (see the *IMPLEMENTATION* section), apparently because device drivers were expected to hang around all the time and burst into action when wanted. In fact, though, there is – and never was – any good reason to regard them as anything but ordinary processes, and as the system has evolved into a multiprogramming system the special position of desk accessories has become an anomaly. In System 7, the distinction has essentially disappeared, and all processes are managed in much the same way[EXE2] : "There are no compelling reasons to create desk accessories for System 7.0". We don't know why they didn't start off with this design from the beginning.

LANGUAGE.

Within the subject called operating systems, processes are about the most lively entities around. While it is processors and devices which directly cause things to happen, they are always instructed by processes; the processes are the components of the system in which decisions are most obviously made and wheels set into motion to carry them out. It is therefore quite difficult to avoid thinking and speaking of processes as though they were people, having desires and needs, making decisions, acquiring resources they need in order to satisfy their desires and carry out their decisions, and otherwise behaving quite like we do.

There is a school of thought in which this anthropomorphism is thought of as undesirable, on the grounds that it is factually wrong, and therefore likely to lead one into further error. This view is certainly correct on the first count; any desires, decisions, or needs have more to do with the plans of the system designer than with the software, and can always (one could hope) be traced back to some design decision. There is also undoubtedly some danger of the second possibility, if only because people might neglect to enquire too closely into the process's motives, regarding the exercise, by analogy with the corresponding exercise with real people, as fairly futile. Any such view is of course silly; a process's reasons for actions are always to be found in its code, and (one could again hope) in its documentation. We might also observe that a significant part of the computer industry is dedicated to presenting false ideas about computers; we have already studied these *system metaphors* (particularly in the chapters *GUI : GRAPHICAL USER INTERFACE* and *ABOUT GRAPHICAL USER INTERFACES*), and seen how they can contribute to making it easier to learn to use a system.

Our view is that a little anthropomorphism does no harm, and can often make for much simpler explanations – provided that the reader bears in mind that it is just as much a part of the system metaphor, and therefore just as illusory, as the desktop. For most people using computers, it is probably quite reasonable to think of computers as having some human attributes as part of the system metaphor; students of operating systems might use the vocabulary, but should decline to be fooled.

We shall not go out of our way to attribute human desires to processes, but will do so occasionally if it seems to make a description easier to understand. We shall expect you to interpret such descriptions appropriately. Just don't start believing that computers have feelings.

COMPARE :

Silberschatz and Galvin[INT4] : Sections 4.1, 4.5.

REFERENCE.

EXE1 : A.H. Veen : "Dataflow machine architecture", *Computing Surveys* **18**, 365 (1986).

EXE2 : *THINK Reference* (Symantec Corporation, 1992).

QUESTIONS

In the history section we saw that multiprogramming was introduced in order to maximise the use of the expensive computer hardware. What are the current reasons for multiprogramming, especially with regard to computer systems designed for individual use?

Has the change in emphasis raised in the previous question caused any difference in the way multiprogramming has been provided by the operating system?

We suggested that "if we want to define processes then we must do so at a level higher than the basic hardware". That's a conclusion which follows from a simple model of a processor. Would it be useful to construct a processor which was conscious of processes ? What advantages would there be to doing so ?

Our decision that threads must be provided with full PCBs was forced upon us because we couldn't tell what a thread was going to do. Is there any reason why we couldn't define threads with built-in constraints which would be easier to manage than full PCBs ? (Perhaps their not-quite-PCBs would only need to accommodate programme counters, for example.)

If we were able to move a running process from one computer to another, what extra information would we need ?
