CLEVER TRICKS WITH PROCESSES

In the previous chapter, we concentrated on the basic transition diagram which describes general, if primitive, process execution. This by no means exhausts the repertoire of useful things to do with processes, and in this chapter we describe a few other interesting techniques.

CHANGING THE PROGRAMME.

You might recall that we said earlier that a process could involve more than one programme. If your system doesn't provide multiprogramming for general use, keeping the PCB and changing the programme is almost the obvious way to work. In that case, as you will always be using exactly one process, all the necessary administration can be done with one PCB. The changeover is simply (well, fairly simply) a matter of stopping one process (see above) and starting another, with the minor simplification that identification information in the PCB can be preserved.

An extension of this idea was used to great effect in Tops10, an earlyish system for interactive programming developed for DEC-10 machines. In this system, a process was in effect identified with a terminal session, so that session control and process control became much the same thing. As you worked at your terminal, running different programmes from time to time, much the same sorts of operation as we have described in the previous chapters went on, but all used the same "PCB". So long as you run only one programme at once, that works quite well. We don't recommend it, because it mixes together two sorts of information (session and process information) which are not logically connected, but it works - and it does embody the notion of continuity, which we saw as essential to the nature of a process, though in this case the continuity is that of the session rather than the nature of the work in progress.

How do you manage if you really do want a new process ? The Tops10 answer is trivially obvious, once you've thought of it : you set up a virtual terminal session, and carry on with that. That works, too, though the confusion of terminal session and process has led you to invent a totally mythical and equally unnecessary device, thereby illustrating the virtue of keeping separate things separate. The implementation was made easier by the quite advanced terminal-handling system incorporated in Tops10, which was close to the final pattern described in the chapter *USING TERMINALS*.

You don't have to rely on a terminal session to justify including several programmes in one process. The analogy of the baker is again instructive. After the cake is baked, it might then be eaten, or sold, or packed, or .... . There is no obvious reason why you should write a single programme to bake and eat the cake, and another to bake and sell it; it makes much more sense to have separate programmes for baking, eating, selling, and so on. Despite that, a sequence such as bake - pack - sell - unpack - eat retains a sense of continuity (particularly if you're the cake), and therefore is at least a candidate for the status of process. If that's a bit fanciful for you, we can point to a set of programmes for matrix manipulation, with the various matrix operations as programmes and the matrices as the cake. It worked, a long time ago, on an IBM1130 computer.

Unfortunately, honesty compels us to admit that the IBM1130 monitor system contained no structure which took any account of what we'd now call a process. The implementation depended on the sharing of the Fortran **common** area which we discussed a long time ago. The **chain** instructions found in many implementations of Basic for small microcomputers are similar, though we don't think that they permitted the direct transfer of data between programmes in a correspondingly simple way. Both these methods were provided as a means of performing extensive sequences of operations on large data structures in small computers, and both preceded the recognition of process structure. Even so, they make the point that the notion of several programmes being active in one process isn't silly.

A quite different way of combining several programmes in one process is to execute them as *subprocesses in a hierarchical process structure. This method was used in the Burroughs MCP system, where every job was represented by a process which set up new processes to run each step of the job. The hierarchy could be continued to an arbitrary depth if required. With such a method, though, there is no sense in which the identity of the process continues through the jobs, and the possibility that one might change the programme without changing the data, as was useful for the matrices, does not arise.*

Perhaps the oddest example is the Unix **exec** function. This is slightly tricky, because it has to replace the programme in which it is executed by a new programme which is specified by the **exec** arguments. The process stays the same, so fields in the PCB which identify it – such as those which identify the process, the parent, the owner – do not change. Things to do with files don't change either. This rather odd behaviour means that the new programme carries on with the old programme's files – and if, as is usual, **exec** is used immediately after **fork**, these files are shared with the parent. This has nothing whatever to do with general principles of process management, but lots to do with how Unix works. Many other fields of the PCB which describe properties of the process rather than define its identity do change.

This is how **exec** works :

- If the specified code file isn't there, or isn't executable, or is inaccessible for security reasons, **exec** returns to the calling programme with a function value which identifies the cause of the failure. This is exceptional : if all goes well, no code following the **exec** call in the original programme will ever be executed.

- Any parameters for the new programme given to **exec** are saved in a temporary buffer. This is necessary, because we are just about to destroy the old programme's memory, which is likely to be the current home of the parameters.

- Currently held memory for stacks, data, and code is released. This is the point of no return; if something goes wrong now the process cannot recover.

- New memory is acquired, with addressing tables constructed as part of the job.

- The new programme's code is loaded.

- Any saved parameters are transferred to the newly acquired memory.

- The PCB's programme counter is set to the entry point of the new programme, and the process is made *ready*.

PROCESS MIGRATION.

In a distributed system, it is also possible, at least in principle, to move a process from one processor to another. (It remains possible in principle even if the processors are of different types, but some principles are harder than others. So far as we know this variant has not been attempted, except in rather special cases where the processors are disguised to make them look the same; assume in the rest of this discussion that we are dealing with identical processors.) As well as being possible in principle, it could be useful in practice in some circumstances. The example usually quoted is that of a process which is using a file stored at a remote processor and finds that the amount of communication with the remote file significantly exceeds the local communication. In that case, it could be cheaper (in money or time) to move the process bodily to the other processor.

Notice that we are discussing the movement of a process which is active; until it is active, we do not necessarily know what it is going to do. If we are aware before starting that the remote file is to be used, it might be sensible to start the process on the remote processor, but that is a much easier operation.

What is involved in moving a running process ? We know in general terms that we must be able to move the PCB and all that is linked to it, and copy or rebuild the structures so that they are semantically the same as the originals even though there might be differences in detail. The difficulty of this task depends on the interactions between the process and other items at local and remote sites. Such interactions are found at several levels.

The most significant are interactions within the memory; if the processor is such that absolute memory addresses are widely used, the transfer is impossible unless either the whole of the process's memory contents can be shifted to identical addresses in the destination processor, or extensive relocation information is available. Fortunately, absolute memory addresses are not very common except in small systems, as forms of address translation are very widely used for virtual memory. In this case, provided that the addressing tables can be set up properly on the remote site there should be no difficulty in continuing processing.

Interactions with files and streams will be affected, but in a distributed system facilities to implement the required access will normally be readily available, and the transformation, though not necessarily trivial, should be possible.

Problems connected with interactions with other processes might be more difficult to resolve. Exchange of messages addressed by process identifier might not work simply if the identifiers are not unique across the whole distributed system; shared memory raises obvious difficulties.

In straightforward cases, though, the technique is effective. The memory contents (code, data, and PCB) must be transferred with obvious modifications, and appropriate addressing tables constructed in the new processor.

GETTING RID OF UNWELCOME PROCESSES.

(This is the murder bit we promised you earlier.) What do we do when a process starts to misbehave ? Perhaps it is behaving strangely or in a tight loop because of a programming error, perhaps it is using resources – memory, disc space, positions in the process table – in such a way as to hinder other work, but in any case it is doing something which interferes with the proper running of the system, and we want to stop it.

In a very simple system, nothing can be done; if multiprogramming is not used, the running process alone determines what happens, and if it goes wrong in any way it will keep on being wrong until there is some sort of external intervention. Fortunately, if multiprogramming is not used, we usually have a solution : we can switch off the machine. That stops the process very effectively in all cases of which we are aware, but it is hard to avoid the feeling that there should be more elegant ways to do it.

With a simple machine, there is no more elegant way, but we can build in something better if we want to – yet again, we appeal to hardware help when software lets us down. Whatever the something is, it must be able to interrupt the running process, and the simplest device is just a button connected to an interrupt line in the processor and set up to execute some suitable code when the interrupt occurs. The safe code to execute is whatever you use to restart the system, because that will certainly restore the system to a safe state – and that's why some computers have BREAK keys on their keyboards, and the Macintosh and "IBM" personal computers provide curious and eminently forgettable key combinations, both of which do pretty well what we've just described.

Notice that if you want to wrest the processor from the control of the rogue process there *must* be some sort of interrupt. Further, the interrupt must be recognised and made effective. There is no shortage of interrupts in most computer systems, but the commonplace interrupt signals are served by small interrupt handling routines which merely deal with the immediate requirements of the interrupt, and then return. That's why typing input to a runaway programme isn't likely to do much good. How, then, does MS-DOS get away with using the <Control-Alt-Delete> key combination from the keyboard as its process-stopping interrupt ? The answer is obvious enough if you think about what's going on – the interrupt procedure must inspect the keyboard input when the interrupt happens, and, if it's the designated key combination, invoke the restart operation instead of just returning.

That does the job, but it's a rather desperate expedient. Whatever was happening is likely to be lost irrevocably, so you might have to repeat a lot of work. Also, there's no chance of conducting a post mortem examination of your programme's (or, too frequently, the operating system's) corpse to find out what went wrong. Can we do better ? Perhaps (for example) it would be sensible to keep a copy of the system state by copying the contents of memory to a disc file before destroying it, so that we could at least try to analyse the evidence if we chose to. Why don't we put a procedure to do that into the code which handles the emergency restart ?

In practice, it isn't so easy. It would work with simple cases such as the never-ending loop, but if – as is often the case – the disorderly process is itself performing input and output operations, then an interrupt from one of the devices it was using can very easily start it up again, and we're no better off than we started – which means back to the off switch. The reset button or special key combination escapes this complication because, in a single process system, interrupts are usually switched off while an interrupt is being dealt with.

That's bad enough even if no one but you is affected, but in a shared system it's a catastrophe. On the other hand, in a shared system we do at least have some machinery which we might be able to use to provide better ways of escaping from the mess. If we can somehow inform the operating system that some process is misbehaving, the system can stop executing the process.

Now in fact "we" – people – rarely inform the operating system directly of anything at all. (Real reset buttons are comparatively rare.) Instead, we send some sort of message through the user interface manager, which is probably another process in the system, and it is this process which has to inform the process managing part of the operating system. This is what we really want, because it might be another process which determines that a process should be stopped, and the emphasis on a process, rather than the special case of the keyboard, as the active entity in requesting the action is therefore more general.

This is the idea behind the common <control-C> keyboard interrupt signal, and the Macintosh's <option-_-esc> (which, significantly, didn't appear until the basic system had evolved to a state where it could handle several processes). These are usually used to stop the process which is currently in charge of the keyboard; to stop other processes, Unix has *signals* which correspond closely to our notion of a message sent by a process. They were introduced primarily as a means of dealing with processes which had in one way or another gone wrong, and are initiated by a supervisor call named, reasonably, **kill( )**. The same call has since been extended to cover a wider range of interactions between processes, and we shall meet it again shortly.

The <control-C> code is interesting because it illustrates the importance of deciding just what keyboard signals should mean. If it really is an emergency exit, it should not be possible to put obstructions in the way; but, in some systems, it has been regarded more as a convenient get-out-of-the-programme signal. In itself, that's an excellent thing to have (it simplifies the system mental model, and we remarked on it, briefly, in the *PEOPLE AND COMPUTERS* chapter), but it requires a rather different sort of treatment. Generally, there might be information held in a complicated programme which should under normal circumstances be preserved before the process is stopped, so it's appropriate after a <control-C> to permit a process to perform tidying-up operations before halting. The system's application programmer interface must therefore include some provision for programmers to catch the signal if they so wish in order to carry out the last rites. Unfortunately, this facility can be abused by people who decide that they don't really want <control-C> to stop their programmes, but rather to return it to some standard state; if that happens – and it does – then the consistency of the interface is impaired, <control-C> is no longer a universal get-out-of-the-programme signal, and much of its usefulness is destroyed.

QUESTIONS

The detailed description of how **exec** works describes the algorithm without paying attention to virtual memory. Describe how **exec** could work on a demand paged virtual memory system.

Consider the trick of applying several Fortran programmes to the same data. Under what circumstances might a version of **exec** which doesn't destroy the process's data be useful ?

Could you provide an implementation of <control-C> with which it would be possible to save useful data but still guarantee that the programme would stop ?
