WHEN SCHEDULING FAILS

Now we have our operating system ready to run, with (almost) all the things a process might need to keep it happy. We know how to make processes, keep track of them as they run, and allocate resources to them as required. Suppose we (in imagination) try it. Assuming that our imagination produces perfect programmes, it will start. It might run for a long time – it could go on running for ever – but it might not. There is (at least) one potential source of trouble which we haven't yet addressed : what if a process cannot proceed because it is waiting for an event that will never happen ?

To make that a little less abstract, think of it in terms of the process needing a resource of some sort which is not available – then the event is the provision of the resource, whatever it might be. The process might be awaiting another page of memory, or the arrival of a signal from another process (that's a fairly abstract resource, but it fits into the pattern), or for access to some hardware device. The process itself is not misbehaving in any way; but if the resource never becomes available for some reason, it will never continue.

WHAT CAUSES A RESOURCE TO BE UNAVAILABLE ?

There are just two reasons why a process which needs a resource can't have it.

- The resource is not available to the system. Perhaps it doesn't exist; or perhaps it's broken or unplugged; or perhaps it's all there, but being maintained. For a physical resource, that might be a terminal complaint; the process wants a paper tape reader, and there isn't one. If there isn't a device of the required type, the condition really is terminal; if there is a device, the process might be able to wait until it becomes available again. For an abstract resource, it might not be terminal : the process might be waiting for a message or other signal from another process. That's what locks and semaphores are about.

  (The solution to the problem of the non-existent physical resource is for the operating system to notice that the requested device doesn't exist, and to respond with some sort of informative message or other appropriate action. This means that the system should know about the available devices and carry out the obvious checks. You might expect that such an obvious thought would have occurred to all operating system designers; so would we, but it doesn't always happen. We shall not discuss this case any further, except to note that it can be turned to advantage : on a rather old system where no synchronising techniques such as semaphores were provided in the application programmer interface, it was found possible to implement a semaphore very conveniently by making processes request a non-existent paper tape reader.)

- The resource is available to the system, but is fully in use. The process needs more memory, but memory is full; it needs a processor, but the processor is busy; it is trying to start a child process, but the process table is full. If the resource is in use, then some other process is using it, so the requesting process must wait until the active process has finished with the resource.

Notice that, in either case, the resource can only become available because of the action of some process. Either another process must release the resource, or, in the case of an abstract resource, another process must generate it.

WHY SHOULD A RESOURCE REMAIN UNAVAILABLE ?

With the exception of the non-existent hardware, those situations are quite normal. This observation emphasises the point that the processes we are discussing are running normally. Nevertheless, we are supposing that for some reason the required resource, despite the fact that it either does exist or, for an abstract resource, could exist, will *never* become available. We can carry the analysis further.

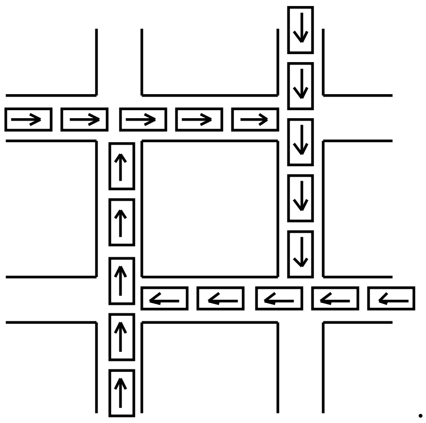
A currently non-existent resource will never become available if no process which can provide it ever does so. This will be so if :

- the potential provider processes are not active : they have all finished, or are never started. There is no general solution for this condition, because it is not necessarily a problem. This is what happens if you do have a paper tape reader, but nobody ever wants to use it. Seen from the system's point of view, there is an interrupt handler ready and waiting forever to deal with the interrupts which never come. Of course, in other cases it might well be a fault, but the operating system can't reach that conclusion unless it has appropriate information. As each process must be considered on its merits, it is probably best to let the designer of the process decide what to do. The operating system can help by providing facilities for *timeout* interrupts, so that a process can wait for some event and request a timeout interval of ten seconds, or five minutes, or whatever is appropriate.

- the potential provider processes are active and running but never execute the code which produces the resource. This condition is quite like that discussed in the preceding paragraph, in that it might be quite normal : any process proving a service might never be called upon to act. (This might even be the desirable case –

consider a process which is intended to handle faults !) Again, the best answer is probably to provide timeout facilities which can be used if they are needed.

- the potential provider processes are active, but are themselves waiting for an unavailable resource. This is the only unambiguously undesirable case; it is typical of the classical deadlock, and we shall discuss it further later. It doesn't only happen in computers, but can be a problem in any system where several entities compete for a limited supply of resources. A very clear example comes from traffic engineering, where it is sometimes called *gridlock* :

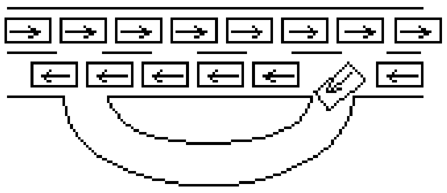
  

  In this example, the competing entities are cars, and the limited resource is the road surface. We'll come back to this illustration later when we discuss the conditions for deadlock; it's interesting to see that they apply just as well to traffic as to processes.

If the resource exists, but remains fully used, we must seek an explanation in the behaviour of the process or processes which are monopolising the resource. Once again, we can distinguish cases.

- In fact, no process is using the resource; a process has finished without properly returning the resource to the system. If the process finished normally, the reason for this fault is incompetence. Even if the process itself didn't properly tidy up its resources, the system should keep track of resource ownership and do the obvious things when a process finishes. There is a little more excuse if the process terminates catastrophically – but not much, as the system should still know which processes own which resources. The moral is that the system must be written so that it *does* have the necessary information, and will use it when necessary. An appropriate programming language can be helpful in ensuring that some resources – particularly those connected with synchronisation and interprocess communication – are properly handled, but can do little about catastrophic failures.

- A process which has the resource will eventually relinquish it, but it will then be allocated to some – presumably better-priority – process, and not to the process which we are discussing. This is *starvation* again (we met it before in the *DISPATCHERS* chapter; the reason for its alternative name *livelock* is perhaps now clearer), and can be a problem with any system of strict and fixed priorities. The answer is to design schedulers so that starvation cannot occur; either relax the *strict* priority requirement, so that resources are allocated to processes of worse priority from time to time, or relax the *fixed* priority assignment, and increase priorities with waiting times. It's interesting to compare this example of livelock in traffic with the deadlock example above :



In this case, the car turning onto the side road can be forced to stay there for ever if the main road traffic doesn't give way – that is, asserts its better priority according to the road code.

- A process which has the resource will never relinquish it, but is running. This might be normal : you wish to make sure that *only* the appropriate operating system device driver deals with the paper tape reader, so you don't want the reader to be allocated to any other process. In that case, the system design is at fault, as the other process should never be able to wait for the device; the system knows that it will never be available, and should comport itself accordingly.

- A process which has the resource will never relinquish it, because it is itself waiting for a resource held by some other process. Deadlock again.

SUMMARY.

Scheduling can fail in many ways, but it is difficult to devise any general scheme to avoid trouble, because no individual process can be identified as the culprit, and a phenomenon which is a failure in some contexts might well be desirable behaviour in others. Where the consequences are unwanted, in some cases the operating system can provide help (timeout facilities), while in others the operating system itself is at fault. There remains a phenomenon which is unambiguously bad; that's deadlock, and is discussed in the next chapter but one.

Scheduling failure is not only difficult to treat – it is difficult to define in any very precise way. We can list some common characteristics of scheduling failure :

- No process is doing anything wrong.
- Some processes are waiting unnecessarily.
- It's a local phenomenon, but can spread.
- The basic problem is resource management :
  - who has what;
  - who wants what;
  - how ownership is transferred.

We have identified two classes of scheduling failure : starvation, which can in principle be cured without harming any process, and deadlock, which can't. We say more about these two topics in the next two chapters.

COMPARE :

Lane and Mooney[INT3] : bits of Chapter 14; section 7.4; Silberschatz and Galvin[INT4] : Chapter 7.

QUESTIONS.

Check that the chapter covers all possible cases. Can you draw any useful finer distinctions ?
