LIFE AND DEATH AMONG THE PROCESSES

In our description so far, we have respectfully stood back from processes and described their parts. This a bit like saying that a car is an assemblage of wheels, engine, and body – true enough (and, so far, similarly incomplete), but it misses out the most important fact. To fill out the picture, we must observe that the whole point of a process – like a car – is to proceed, and in this chapter we shall address the life cycle of a process.

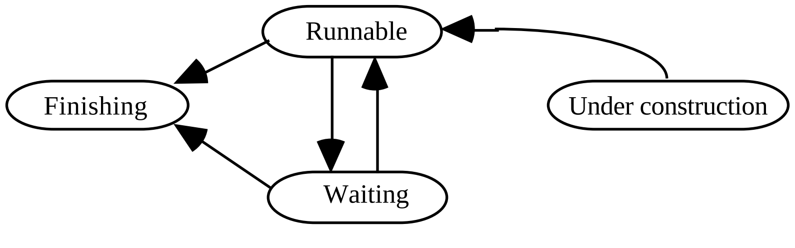
As is not uncommon, we have to begin by defining some terms, and the first is the name of a property of processes which corresponds closely to phases of a person's life. Indeed, the conditions we shall discuss can be thought of as analogous to someone's birth, waking life, sleeping life, and death. We shall follow common usage and call it the *state* of the process, though in fact it is only one part of the set of properties of a process which one might regard as defining a more compendious notion of the state. Later, we shall call this broader set of properties the process's *disposition*.

The state of a process can be roughly defined as what it is doing at the moment, and this is the sense in which we discuss it here. A rather more precise notion of state includes a statement of what the process is going to do next. That gives more detailed information than what it's doing now, because in many systems most processes are doing nothing for almost all the time, so most of their states are of the form "waiting for ...". For example, a process may be running, or waiting to run, or waiting for some resource to become available.

THE PROCESS STATE TRANSITION DIAGRAM.

A process goes through several stages in the course of its existence. From the point of view of getting work done, the most important state is (obviously) when the process is actually running. Or is it ? That rather depends on what sort of process we consider. Many processes which deal with input and output are intended to spend most of their time just waiting for something to happen; while it is certainly true that we expect them to work correctly when they burst into action, one could argue that it's more important that they should be waiting, ready to act, all the time. We might also think of processes intended to deal with faults – which we hope will never run at all ! Nevertheless, we want them to be there, ready for service if it should be required. In any case, though, the actively running time is usually only a small percentage of the time the process is in existence.

The usual way to describe these state types and the way they fit together is to present a *state transition diagram*. There are many ways to draw such a diagram; different operating systems can use slightly different sorts of state, and can give them different names, and one can draw finer and finer distinctions between slightly different sorts of process behaviour thereby inventing many more state types. The diagram below shows a fairly typical system described in not very great detail.



Now we shall discuss each of the state types shown on the diagram.

Under construction.

While often not strictly necessary, it is possible, and usual, for the operating system to allocate various resources to the process before it starts running. Allocating resources at this point is useful to prevent conflicts between processes for resources, which can stop the processing (called *deadlock*, and discussed in our final section), but most interactive systems allow processes to request resources while they run. By this means, routines to set up the process are simplified.

One important resource deserves special mention. It is usually necessary to allocate some memory to the process and to load some or all of the programme code ready for execution. It is common to load at least the code and to reserve space for some memory requirements; for example, if an execution stack is required, this must be set up from the start. Just how much is needed depends on the design of the memory management system and the requirements of the programme. The traditional method, still adopted by some systems, is simply to allocate an area which should be big enough for all reasonable variants of programme behaviour and then leave the programme to fend for itself.

It is not strictly logically necessary to allocate memory at the start because of the miraculous properties of virtual memory. In theory, one might hope that, provided that the code is on the disc to begin with, the system could rely on the virtual memory machinery to fetch the required parts for itself when necessary – but several conditions must be satisfied if this is to be possible. The most important of these is that the correct disc addresses must be place in the initial addressing table so that the code can be found when it is required. The code origin alone is not sufficient; a branch might lead to any part of the code at any time, so the complete table must be set up. A second condition, not strictly necessary but certainly expedient, is that the layout of the code on the disc must be compatible with the requirements of the virtual memory system. It must be easy to read any page or segment when required, so the layout must be just as it is in memory, and disc sector boundaries must come in convenient places. If these conditions are not satisfied, then it is hard to avoid reading all the code into memory so that the virtual memory system can look after (and ensure the correctness of) the details of writing it back to disc.

However a new process is constructed, it is extremely important that it should be set up in conformity with the operating system conventions – if not, the system is likely to lose control. While that is fairly obvious, in practice it isn't difficult to get it wrong. A very easy way to go wrong is to try to go it alone – to set up the system how you know it has to be, but in some special and convenient way. The result is that there are two ways to (using the example above) set up the addressing table, one used when a process starts and one used while it's running. Now you decide to improve the memory management system, and forget to patch one of the two procedures. We'll say this again when we discuss starting up the whole system (*STARTING AND STOPPING*), but we think it sufficiently important to say twice : avoid special methods as far as you can; do the minimum in the way of special things, and use the standard routines whenever possible.

Runnable

Once resources have been allocated to allow the process to run, the process can now be passed on to the care of another part of the operating system. This is the first important change in the life of the process, to the *runnable* state. This promotion is often called *scheduling*; it is one example of a much broader set of scheduling functions performed by the operating system, all of which can be described as deciding what happens when. Scheduling is a very important activity in managing the operation of the system, and we discuss it, including more detail of what happens to processes with *runnable* state, in the *MANAGEMENT* section; here we concentrate on the operations on the process which must be carried out in order to get it into a *runnable* state, and maintain it there until it finishes.

That a process is *runnable* does not mean it is *running*. Even in a nominally single-process system, the process could be interrupted, but in a multiprogramming system it is merely one of the set of processes from which the operating system chooses in order to select one process to run. There are several algorithms which can be used to make this selection, and we discuss these too in the *MANAGEMENT* section. Being *runnable* does mean that the process can run whenever a processor becomes available; it is not waiting for any resource except a processor.

The *runnable* state is commonly seen as having two component state types, *running* and *ready*. In a multiprogramming system, almost all of the *runnable* processes are *ready*. Putting that another way, hardly any of the processes are running – for only one thread per processor can be running.

Waiting

It is quite likely that, very soon after the process begins running, it will make a request for a resource which is not immediately available. It might be a request for memory, or a requirement for data, perhaps resulting in a wait for input from the keyboard, or for a file to be opened.

When this happens the process relinquishes its processor. If there is only one processor, it has to, as the system will require the processor to do whatever must be done to acquire the resource and give it to the process. If, on the other hand, there are many processors, then it would be possible in principle to stop the processor allocated to the requesting process, and restart it when the resource becomes available – except that modern processors rarely have **stop** instructions, and a multiprocessor system is highly likely to have a multiprogramming operating system which will in any case try to use the waiting time for some other process.

When we say the "process relinquishes its processor" that isn't really how it appears to the process - indeed, the process doesn't even notice. Every request for a resource is sent to the operating system. (Recall that the operating system is still a resource manager, even though we've expanded the original definition.) If the operating system discovers that the resource is not available, it does whatever it must do to record the request and ensure that it will receive proper attention, then changes the state of the *running* process to *waiting*, and (in a multiprogramming system) gives the processor to a *ready* process.

What is the process doing while it is not active ? We have called the state *waiting*, but it is not waiting in any human sense. When we wait, we are conscious of time passing – we *know* that we are waiting. For the process, though, time is measured in processor cycles, and if it is using no processor cycles then for it there is no time. It is very important that this should be so, for it is our guarantee that we can achieve a functional multiprogramming system, which would not be so if a process could tell how it was being treated, and therefore adapt its behaviour to compensate. In practice, we often speak of *waiting* processes as "asleep", and that is perhaps a rather closer metaphor than "waiting".

It is common to distinguish two sorts of *waiting* state, usually called *blocked* and *suspended*. A process is blocked when it requests some resource which is not immediately available, so has to wait for it. A process is suspended when it is waiting for an event which has nothing directly to do with the process itself. Suspension is really an operation for use when scheduling processes for the good of the operating system rather than something brought upon a process by its own actions; it is usually inflicted either by another process or by someone controlling the system, perhaps to reduce the processing load or to keep a process for investigation if something has gone wrong.

Finishing

Eventually, most processes come to an end. (The exceptions are system processes set up as services of one sort or another – such as processes which tend devices, or wait for faults, or desk accessories.) Apart from the possibility of a system crash, there are two main ways in which this can happen : the process can come to its proper conclusion by executing its last instruction, or the process can be destroyed by the system. There are many reasons why an operating system might wish to destroy a process : the process might be waiting for an unobtainable resource, it might have used up its time allowance, it might have tried to execute a privileged instruction or use memory to which it does not have access rights, and so on. The normal end corresponds to the transition of a *runnable* process which is in fact *running* to the *finishing* state in the diagram; the forced end must be forced by something else, so – on a single-processor system – corresponds to the transition of any process which is not *running* to the *finishing* state.

Before the operating system can take such summary action, it must somehow find out that the action is necessary. In the case of processes which attempt to use a resource which they are not entitled to use, the operating system discovers the misbehaviour when the process requests the resource or attempts to use a privileged instruction; depending on the system design and the nature of the fault, it might then return an error indication to the process or terminate it forthwith.

Some of the other conditions are not so easy to spot. How can the operating system tell whether or not a resource which is not immediately available will, in the foreseeable future, become available ? Certainty is only possible if the operating system either knows that the resource really doesn't exist or has supervised its permanent removal from the system. (For example, a process might wait for a message from another process which does not exist.) Without reliable information, our interactive operating system has to presume that a requested resource will become available, or it might stop some process which could legitimately continue to wait. To avoid infinite waits, it is possible to prescribe an upper limit to the waiting time (a *timeout*) for a resource, and to remove any process which has waited for longer.

A closely related problem is that of *deadlock*, where two or more processes form a cycle waiting for resources which will never become available because the processes are all holding at least one resource which another in the cycle requires. We discuss this topic further in the *MANAGEMENT* section.

QUESTIONS

Consider carefully the suggestion that it should be possible to start a process without allocating memory, by relying on the virtual memory system. Just what would have to be done to make it work ? Would it be worth while ? Would your answer change in a system where a large (say, 64-bit) address space made it possible to store complex structures directly to files ?
