PROCESSES IN ACTION

Now we have the means to keep records of the parts of a process and what they are doing, we can use this to organise the operating system's control over processes which are running in the system.

PROCESS TABLES.

The PCBs have to be accessible to the operating system so that it can use or change the information whenever it is needed. There is no pressing reason why the process itself should have direct access to the PCB (though indirect access to some of the components mediated by supervisor calls might well be useful), and there are several reasons to do with protection and security why it shouldn't, so the PCB can reasonably be kept in the system memory. As we usually want quick access to any PCB at any time, we need an array; this is what we called the *process table*. Once you build a multiprogramming systems, there is usually no logical limit on the number of processes which it can handle, but most operating systems impose a more or less arbitrary limit and declare a fixed array somewhere of appropriate size – usually chosen to be significantly larger than might be reasonably required in any conceivable practical circumstances. This array might be the process table itself, or it might be something amounting to an array of pointers to separate PCBs. Particularly if the PCBs are large, or if the size of the array is unrealistically high, the pointer technique might significantly reduce the memory required for the process table.

The idea of the process table makes sense of the notion of identifying a PCB by an "index or pointer" which we introduced earlier; a PCB in the process table can obviously be identified by giving its array index. There are at least three reasons why we want to distinguish between table indices and the names by which we identify processes.

- The indices might not identify processes. Unless the table is full, some of its entries will not describe processes, and will therefore not be PCBs. We still need the indices, if only to maintain a free list for the table, but we certainly can't just assume that every index identifies a process.

- There is no deep reason why processes shouldn't move from one position in a process table to another. Indeed, in a distributed system in which process migration is allowed, moving from table to table is specifically implemented. Similarly, a system which allows for very many long-lived processes might be designed to move dormant processes out of the process table entirely, and they are unlikely to acquire the same table slot when they are revived.

- The use of an index is also unsatisfactory if we wish to preserve access to information about a process after it has finished – as we might well wish if we record information about its activities in a system log file. For such purposes we need names which will not recur for quite a long time, or it becomes hard to decode the log.

Notice, by the way, how a process as seen by the operating system is quite a bit more complicated than it appears to someone using the system. It has many components : the PCB, the memory tables, the actual contents of the one or many memory areas which it uses, the states of its files, and so on. Of course, that's why we had to invent the PCB, which acts as a directory through which we can find all the other bits when we need them. There's an interesting parallel with the way we had to extend the idea of a file to include file attributes, file information block, and perhaps several distinct data areas used for different purposes. It's not surprising that object-oriented systems are becoming popular !

THE PROCESS TABLE AND THE STATE TRANSITION DIAGRAM.

A good way to see how the process table is used is to follow a process through the state transition diagram, observing how the information in the process table is used and changed as the computation proceeds.

Under construction.

To construct a process, the operating system must allocate a vacant slot in the process table and build it up into a PCB by filling in the fields as appropriate. We've described the procedure in outline in the chapter *STARTING A PROGRAMME*. Inspect the list of PCB items above : in all (comprehensible) cases there are reasonably obvious initial values which can be set up – including the state, which is *Under construction*. (The operation is often called "process creation", which is a rather pretentious term for a rather modest operation. Fisher and Paykel don't claim to "create" washing machines.) If any initial resources are provided for the process, the initial settings must be chosen to include them.

Runnable

Once the bare bones of the process are ready, its state is made *runnable* (strictly the state becomes *ready*). This sounds rather like tipping a fledgling out of the nest and hoping that it will learn to fly before it hits the ground, but in fact it's all part of the principle of using existing system components as far as possible. The new process will almost certainly need more resources, but we want it to acquire these resources using the ordinary system procedures, and it can only do so if it is *running*. It is therefore made *runnable*, so that it can be run when the processor becomes available, and can then begin to issue requests for the other resources it needs.

So far as the process table is concerned, the state field in the PCB for a *runnable* process is set to *ready*, and the PCB is usually appended to a queue of *ready* processes. When its turn for processing arrives, it will be removed from the queue and executed.

Waiting

The assumption of a single *waiting* state is an oversimplification, as the reason for the process's idleness is important. In *LIFE AND DEATH AMONG THE PROCESSES* we distinguished between blocked and suspended processes, and we can draw further distinctions according to the reason for blocking – in effect, there are different sorts of *waiting* state, and it makes sense to use different levels of waiting. Consider the difference between waiting for a file from a local disc to be opened and waiting for the opening of a file on a magnetic tape, which has to be loaded by the computer operator before it can be read. Opening the disc file will take some time in processor terms, but the overall time should be well within one second; in contrast, waiting for the tape will take so long that it is sensible to relinquish other resources which the process has in the meantime. A common means of dealing with this variety is to set the state of the PCB to *waiting*, and to attach the PCB to one of several *waiting* queues according to the resource which it requires.

What happens when the resource becomes available ? The operating system must make sure that it is used by a process which wants it. If only one process is waiting, that's easy enough; the process is woken up, any housekeeping needed to attach the resource to the process is performed, and then the process is made *ready* again. If there are several waiting processes a decision must be made. The obvious decision is to organise the linked list associated with the resource as a queue, and to give the resource to the process at the head of the queue. This is a common method, but doesn't take account of processes' priorities. A method which does notice priorities, used by Unix, is to return all the processes waiting for the available resource to the *runnable* state, setting all their states to *ready*. The priorities will be taken into account in deciding which process to run; that one gets the resource, and the other reawakened processes will be returned to sleep.

Finishing

Every well behaved process ends by committing suicide. (It's also possible for the system to stop processes before they come to their ends, but now we're discussing the normal behaviour. Murder comes later.) To do so, the process makes an appropriate system call. If there is provision for returning a result code to the system, this might be passed as an argument. You might recall that we saw a need for such information (in the chapter *CONTROLLING THE COMPUTER SYSTEM*) in order to implement satisfactory job control procedures.

Once it is decided, for whatever reason, that a process must be removed, what happens ? Removing a process has to be done very carefully. Recall the complexity of the PCB which we saw earlier. It contains a lot of information in its own right, and also has links to the memory management system, the file system, other processes, and many other entities. Every one of these links must be tidied up in some appropriate way, or we will end up with an inconsistent system – often with some resources which appear to be in use by processes long dead. The operating system must therefore make sure that all resources which have been allocated to the process are returned, otherwise the system will gradually run out of resources. These resources which are directly identified in the PCB can usually be dealt with more or less straightforwardly; for those which are known elsewhere, it might be necessary to scan the appropriate data structures to see whether resources should be freed. To illustrate what is needed, here are a few notes on some particular operations which have to be carried out.

All **open files** should be closed, however the process left them. This is usually more than a matter of convenience, for (as we saw in the *FILES IN THE PROGRAMME* chapter) the state of a file on the disc is not necessarily always up to date with what the programme has done to it, as there are changes in memory buffers and file information blocks which have not yet been stored on the disc. If the file is not properly closed, these changes might be lost. As well as that, other people might be denied access to the file because it appears to be in use, and the device associated with the file will also appear to be in use.

Accounting information must be brought up to date. Usually people have to pay for the amount of computer time (and other resources) they use. The payment might be logical rather than capital (i.e. a reduction in quota), or it might be non-existent, but even in this case we might still want to keep track of who is consuming how much of what resources.

Relations should be notified where appropriate. If a process completes its work or is removed from the system while it is doing some work for another process, the survivor must be notified. In this case the result or output produced by the *finishing* process can be understood as a resource for the related process. Just as processes wait for information from devices they can wait for information from other processes.

Only after cleaning up completely should the PCB be freed. By this time, of course, it is really no longer a PCB – it's just a vacant slot in the process table, and can conveniently be linked into a list of vacant slots or otherwise identified as available for use.

TWO WAYS TO BUILD A PROCESS CONTROL BLOCK.

We have described how a process table might be managed in a simple and straightforward way for a simple and straightforward operating system, but there is no law which says that you must follow that pattern. Many systems deviate from it to some degree, usually because they have some good reason to do so – they might be designed for special purposes in which unusual factors must be taken into account, or they might have more complex file structures or process structures which require special treatment. Generally, though, the essential pattern of process management remains the same even if the details change, and the model we have described is a good starting point for understanding a new system.

To give some idea of the possible variety, we shall describe two rather different ways to build a PCB for a new process. Our first description is fairly brief, because we've already given most of the information in the chapter *STARTING A PROGRAMME*. The procedure described in outline there is the obvious, and usual, way to build the PCB. It begins with a blank process table entry (not strictly a PCB yet), then acquires the various components of the process structure from wherever is appropriate, and makes entries in the nascent PCB in the straightforward way. Once all the components have been assembled, and the PCB is complete, it can be attached to the *ready* queue and left to look after itself.

Some information is required by the system before it can perform the job. For the minimal example given, it must be told the name of the code file and presented with an argument list. Such a facility might be made available as an API component called by a procedure call something like

start(<codefilename>, <argument structure>)

Early systems didn't provide means for processes to start new ones, but as multiprogramming has become better understood and its benefits have been perceived such facilities have become more common. The procedure might reasonably return a result code to show what happened when it was executed. Should it also return a result from the new process when it finishes ? That's a design decision - if that's what you want, you have to build your system to do it.

And now, for something completely different, we describe how processes are managed in the Unix system[EXE4]. Unix avoids some of the hard work of process building in a way which is curious, interesting, and quite peculiar, though rather elegant, and we shall therefore describe it in a little detail.

In Unix processes are always constructed by copying other processes, and any process which wishes to start another may do so. (The first part of that statement is obviously untrue, but it's true almost all the time. We shall deal with the other bit in the chapter *STARTING AND STOPPING*, which comes right at the end.) When a process wants to create another process, it asks the operating system, through a system call named **fork**, to make it into twins. The **fork** procedure has no parameters; like other Unix system calls, it is a function, and returns a value which gives information to the process to which the call returns. (We choose our words with some care : read on.) This is how it works:

First, a vacant entry in the process table is acquired, and new process identifier is allocated and stored in the PCB. The identifier of the original process is copied from its PCB and written into the new PCB as the parent's identity. The state of the new process is set to *being created*, which we don't much like, but Unix is like that. The original process is one of the twins; to set up the other, while permitting the two twins to run independently, we have to make copies of anything belonging to the parent that can change and attach it to the child's PCB, but the twins can share anything guaranteed to be constant. (We apologise for the somewhat bizarre reference to twins as parent and child, but it seems to be the easiest way to describe the operation. You might recall our comments on anthropomorphism.)

In practice, most of the components must be copied. These are the PCB contents, including the processor registers, and the process's data areas, including the process stack, with appropriate changes to addresses in the new PCB in both cases. The code is usually shared, so its address in the PCB need not be changed; if for any reason the code cannot be shared, it must be copied too. Files are not copied; the two processes share the same open files, which can raise complications if they aren't careful but also has certain advantages. Outside the PCBs, other details must be changed to keep the system consistent : for example, the usage counters of files open in the parent process and regions of shared memory are increased.

The new process state is set to *ready*, which is one of the *runnable* status types, and the operation is almost complete. We have two processes where we originally had one; they are, so far, functionally identical, and that won't do at all. It won't do, because

if neither process knows whether it is parent or child, we have no systematic way to make them do different things, which is usually what we want. We emphasise that they really are effectively identical; if we simply let them run without doing something clever, they will continue to be identical as well as they can, in the process messing up each others files (which covers just about everything in Unix) by doing everything twice, more or less, making matters worse and worse as they get further and further out of step. This was not quite what we had in mind.

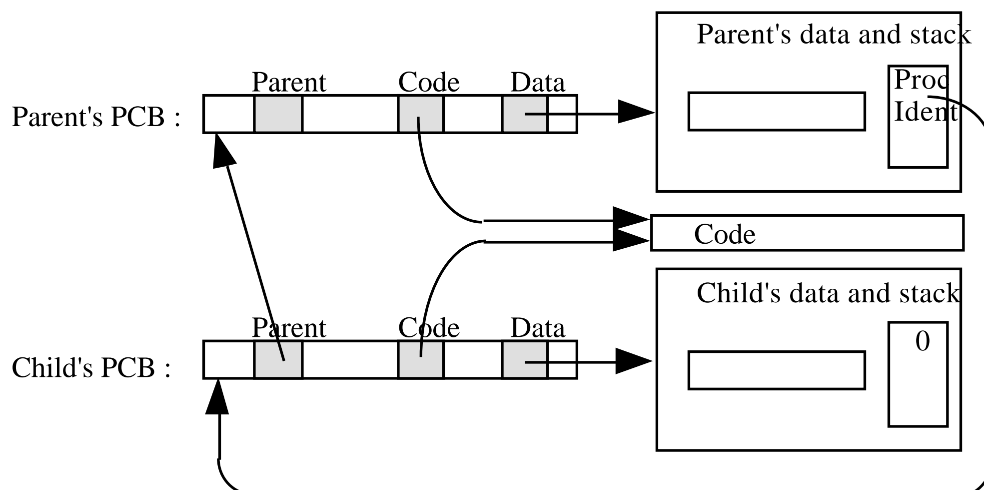
Pause a moment, and try to think how you would solve the problem. If you already know the answer, forget it; we would like to impress upon you that the solution, simple though it might be, really is clever. Two identical processes must somehow become unidentical. They have no easy way to talk to each other (ways exist, but they aren't much use for the task in hand) – and, even if they could converse, what would they say ? One might hope to rely on one running before the other, but on a multiprocessor machine even that is not guaranteed Each of them believes itself to be the original process, having just called **fork** and now proceeding after the call. How do they know which is which ?

The answer uses the one remaining tool which we can use for the job : we have not yet executed the procedure returns. When doing so, the system - which *does* know that there are two of them - sets different returned values from the **fork** function in the two processes. One receives zero, and the other receives the process identifier of the other. As the child already knows who its parent is (because it's in the PCB) it is sensible to return zero to the child, and the child's PCB to the parent. Notice that the situation is unsymmetrical; the child can always identify its parent by using a system call, whereas if the parent loses its child process identifier – which, we must admit, is very careless – there is no simple way to find it.

The result of this operation is a pair of essentially identical processes, each a continuation of the process which made the request. One of these is regarded as the original process, and one as the new one. Initially, both are executing the same – the original – programme, and that continues after return from the **fork** call. Both processes have just executed something amounting to this sequence :

```
Call fork and receive the value.
If the value is negative
then something went wrong
else if the value is zero
     then I am the child
     else I am the parent, and the value identifies the
                child.
```

We can think of the result like this :



Notice that the same **fork** code will work for all cases, and if the italicised parts are replaced by instructions which say what to do in the cases described will behave just as required. Notice, too, that this is only the first step; even though the processes now know who's who, and have their own data areas in memory, they still have to sort out any potential tangles which might result from their common ownership of files. We shall not address these problems here, except insofar as they concern the job of starting a new programme, which is the most common requirement in the operating system. In this case, the new process uses a second system call – **exec** – to ask the operating system to make the required change. The **exec** function ends a programme, not a process, and it is an excellent example of the distinction between programme and process. We'll discuss it in the next chapter.

REFERENCES.

EXE4 : M.J. Bach : *The design of the Unix operating system* (Prentice-Hall, 1986).

QUESTIONS

What should happen if there is no PCB available when a request comes to the operating system to create a new process ?

How can the operating system determine when a process has used up its processor time allowance ?

We wondered whether a process should return a result to its parent when it ends. Should it ? What sort of structures will you have to put into the operating system if that's what you want ? What will happen if the parent has finished ?

There is no guarantee that a programme will execute code to clean up its resources before finishing. How does the operating system take over and clean up before the process is finished with ?

We mention that the traditional **fork** is particularly inefficient in virtual memory systems because of the disruptions caused by page faults when acquiring memory. What memory has to be acquired? Even if there is enough real memory available for the new process, disruptions are still likely, explain how.

How would forced termination in a Unix system differ from the normal termination procedure outlined ?