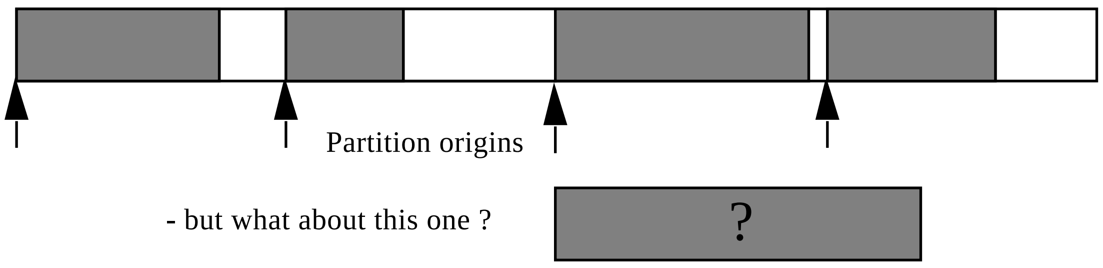
MEMORY TRICKS

Now we must address the first question raised at the beginning of the preceding chapter : how can we fit several programmes into memory at once ? There's another question which falls in the same area, and which we noticed at the end of the preceding chapter : how can we isolate programmes in memory so that they can't interfere with each other ?

PARTITIONS.

One way to get several jobs into memory at once is a fairly straightforward extension of the spool system's memory management. Instead of having just one memory area into which we can put a programme, we have several, which we call *partitions*. The boundaries are still arbitrary, because there is still no help from the hardware, but they are fixed, so that we can tell the linker and loader what range of addresses to use when building the programme code – for example, in a system providing eight 64K partitions the permissible address range for a programme running in the third partition is 128K to 192K-1. Of course, once your programme was linked and relocation completed within that address range, it would only run in the third partition; a relocating loader could help, but at the cost of extra time, and the usual practice was to allocate a job its own partition, which would change rarely, if ever.

The diagram below illustrates the idea. We've shown a memory with four partitions, and five blocks of code (shaded areas) intended to fit in.



Two complications are immediately obvious, both consequences of the system's rigidity. While small programmes can be executed without difficulty, they leave a lot of memory unused, so much space might be wasted. For large programmes, it is possible to twiddle the details so that partitions may be combined together when a large area of memory is required, but any such irregularity is bound to interfere with the smooth running of the system. Another difficulty is less obvious, as it is connected with the execution rather than the geography. We remarked that the memory management, such as it is, for partitioning is much the same as that used to hide the input and output software in a spool system; unfortunately, there is no similar analogy for controlling the programme execution. In a spool system, one process is normally running, and the processor can be caused to redirect its attention to the code managing a device by an interrupt from the device. If the processor in a partitioned system is executing the code in partition 1, what is there to redirect its attention to the code in partition 2 ?

To make this work, we have to have cleverer system software, which will somehow find out when the current active programme cannot immediately proceed (or has completed its activities), and will then save the environment somewhere and restart another foreground programme. The system must also keep track of which partitions are in use and which are available for new programmes. The details are not particularly significant at the moment; the important point is that for the first time we need system software explicitly to manage process switching.

This is an example of what quite typically happens if we try to make a system more efficient; the method we use introduces some new feature into the system, and this in turn imposes additional demands which increase the complexity still further. In this case, the new feature is *multiprogramming* – the processor is shared by two or more programmes. The additional complexity which we have just addressed is the need for process control, but it is one of many. Another consequence of sharing the system between programmes is the possibility of interference between them; there's nothing (yet) special about the computer, so there's nothing to prevent a programme in partition 1 using an address in partition 2 (or anywhere else), and no one has yet found a way to guarantee that programmes will be perfectly correct. If this happens, we can no longer guarantee a functional system; on a more mundane level, someone else might mess up your programme. We'll describe ways of dealing with the problem later, but the point is that such problems do appear whenever the system becomes more complex.

OVERLAYS.

What, then, can you do if your programme is too big to fit into your partition ? (– or your computer, for that matter ?) Not all is lost[*] : very commonly you don't need all your programme in memory at once, so the trick is to load into memory only the bit you need at the moment.

In an *overlay* system, part of the programme code is kept on the disc and part (including, obviously, the part that's running now) in memory. The major difference between overlay methods and the virtual memory systems which came later is that in virtual memory the transfer between memory and disc is managed by the operating system, but in an overlay system the responsibility for memory management lies with the programme itself. (The CHAIN instruction found in some small systems is another approach to the problem; a programme can use CHAIN to run another programme, but in that case there is no "main programme" left to exercise overall guidance.)

To give an example of the overlay technique : consider a simple programme composed of a main programme M and two subroutines A and B. Here's a picture of it in something-like-Pascal :

```
procedure A;
begin
    do_things_not_involving_B;
end;

procedure B;
begin
    do_things_not_involving_A;
end;

{ Main programme.           }
do_things;
A;
do_other_things;
B;
do_yet_more_things;
end.
```

Suppose M, A, and B each occupy 5 units of memory; then (neglecting any overheads) the programme can run in a computer with between 10 and 14 units of memory provided that only two parts of the programme need be in memory at any moment. If we assume, reasonably, that M has to be there all the time to keep control, then the condition is that we never need both A and B resident in memory at the same time. The code for M, A, and B is kept on the disc; M's code is brought into memory to start things off, but then M itself can load A or B into the vacant area of memory as it requires. The method works quite well – because the required separation into more or less independent subroutines is a realistic assumption. That's why structured programming techniques work.

Now we must address the other part of the question : how do we keep track of what is where ? Clearly, we can no longer simply call, say, procedure B. Instead, we must first ask whether or not B is present in memory; if not, B must be loaded. Then B can be entered in the usual way.

In practice, the ordinary programmer should not have to worry about the details, which are handled by software associated with the linker. The programmer does have to say which procedures are to be handled as overlays, because the decision must be based

[*] : This has nothing whatever to do with operating systems, but you may care to ponder the difference between the phrase "*Not all is lost*" and the more common form "*All is not lost*", and to decide which is the more appropriate to the circumstances in which the phrase is conventionally used.

on an understanding of the flow of control through the programme, and that information is not necessarily available to the linker.

The information is available to the compiler; should it be made available to the system as a whole, so that memory management strategies can be made more precise ? That's a matter to be decided when designing the operating system. If the information is required, it must be laid down as a system standard requirement. To our knowledge, no system has stipulated that compiler code files should contain information of this sort.

There is a further difficulty. Our argument has been based on programme execution, and has concentrated on memory requirements of executing code. What about data ? If there are data areas which are used by many different parts of the programme, the pattern of memory use is greatly complicated. (It is perhaps relevant that, when overlays were first used, the idea that data areas could be thought of independently of programme units (a Fortran term) was not widespread; it wasn't easy to do in Fortran or Cobol, few people used Algol, and, apart from a few very specialised areas, that was all.) The same sort of question arises if data are kept in a programme stack, which is also to some extent independent of the code. The overlay technique does not work so well for systems which use these more flexible models of memory.

Overlays are rarely, if ever, used today, having been superseded by virtual memory methods. They remain of academic interest because they show very clearly how locality of reference can be exploited, and how it is connected with programme structure.

QUESTIONS.

One difference between the different memory management techniques is in the binding time for addresses (the time at which the actual address is determined). Compare the different systems.

Invent a programme with a main segment and several procedures, calling each other in plausible ways. (You needn't write any code.) Allocate sizes to the procedures. Now consider how to execute the programme in a memory that's too small using overlay techniques. It's usually supposed that overlays include one or several complete procedures, and that they are only read from the disc (so you can't store current values of variables for a while to do something else). How can the necessary operations be managed ? Can you invent an algorithm to allocate the overlay areas automatically ? What information do you need to do that ?
