MEMORY MANAGEMENT : THE PROCESSES' VIEW

We could probably finish off this chapter in about ten lines, but instead we shall spend the ten (or a few more) lines on telling you why the chapter is much longer.

The chapter could be very short, because the view which we shall discuss is very simple. We have decided to restrict the main thread of our discussion to the flat memory model (we reserve the right to digress if it seems like a good idea), so any process's view of memory contains only a set of integers starting at N and increasing in steps of 1 up to M. In the interests of consistency, it's obviously best to make N zero; and, if we talk about resizing, M is variable. Its view of memory management is similarly simple, and is restricted to the **lowerlimit**, **upperlimit**, and perhaps **resize** functions we discussed earlier. We could find a few more things to say, but that's the end of the discussion.

The chapter will not be very short, because the subject of the course is not processes, but operating systems. We shall therefore discuss the means used by the operating system to satisfy the processes' requirements for a view of a flat memory. (Perhaps the chapter should be renamed "MEMORY MANAGEMENT : THE SYSTEM'S VIEW OF THE PROCESSES' VIEW". And perhaps not.)

One advantage of the extended chapter title which we have just rejected is that it is more precise; in particular, it emphasises that we are concerned only with the processes' external interaction with the system, and not the memory models seen by the programmer. The programming language used to write the programme might be expressed in terms of stacks, lists, objects, or whatever – but our assumption about the underlying flat memory model means that the flat model is all we must support.

With these considerations in mind, then, we embark on our discussion. You will observe that it is rather more than ten lines in length.

A SIMPLE MEMORY SERVICE FOR A SINGLE PROCESS.

The fundamental system action for a process is to provide it with a chunk of memory in which to put its code and data. We reviewed the basic functions in the *MEMORY MODELS* chapter, where we distinguished between the sorts of service we might expect to provide in systems designed to run just one process and those for running several processes at the same time. In fact, this is a rather low-level view of the memory, which is unsatisfactory in practice; we would prefer that a process should not need to know any more details than it has to about what sort of environment surrounds it. Also in fact, unfortunately, both patterns are still found in operating systems, so it's impossible to define a uniform interface which will work with all systems. That's a specific example of the reasons behind attempts to define standard application programme interfaces, such as Posix, and we hope that they will become widespread, but history suggests that they won't be observed by people developing new sorts of system if they're inconvenient. After all, we knew how to build quite good operating systems before microcomputers came along, but many of the lessons we'd learnt were ignored by the microcomputer system developers. But we digress.

Perhaps the simplest imaginable case is the *single contiguous allocation* technique, in which a process is provided with a single area of flat memory. This can be done in two ways, depending on what part of the system defines the size of the area. If the size is defined by the system, then a method based on the use of the **lowerlimit** and **upperlimit**, functions is appropriate, while if the process defines the size required we would expect some form of **get( )** function to be provided.

The system-defined techniques were those used in the early days of computing, and we have mentioned them in the *HISTORY* section. Many small systems still use variants of the **upperlimit** and **lowerlimit** method. From our top-down design viewpoint, **get( )** is clearly the preferred method, because it emphasises the process's requirements rather than the system's limitations – and, for once, development is going in our direction ! Observe, though, that in the single contiguous allocation case, it is only **get( )** in a metaphorical sense; the process defines the memory size required, but cannot itself execute a **get( )** function as it owns no memory in which to execute it until after the execution.

The use of **upperlimit** and **lowerlimit** raises other difficulties. Even with a very simple system, the origin of the area is unlikely to be at address 0, as it is very common for the first 2^n memory cells to have some special hardware-dependent characteristic such as a compact or fast addressing mode which makes them a valuable resource, not to be wasted on mere programmes. Such an area is often called by some name such as the *base page*, and reserved to give ready access to operating system facilities. There might also be the resident monitor, which must be kept somewhere. The high end of memory is also a useful repository for things which must be kept out of the way. Therefore, unless the computer hardware includes base and limit registers or something equivalent, **upperlimit** and **lowerlimit** (the starting address of the allocated memory area) must be made available. The values might be defined as fixed numbers, or they might be made available by functions as we described earlier. The use of fixed values is much more convenient, for then they can be built into compilers or loaders so that most programmers don't need to worry about the problem; unfortunately, limits fixed by decree at an early stage in the development of a computer system will almost certainly turn out to be mistakes after a few years* of evolution, by which time it will be quite impossible to change because vast quantities of running software will depend on the original values. (Moral : if you design computer systems, avoid unchangeable constants at all costs.) Variable limits are a little more complicated, for then one must use relocatable code with a relocating loader, though that's really not very difficult to manage.

It seems, then, that while these basic functions work in principle, if they are implemented in a simple computer they are both inflexible and dangerous. They are *inflexible* because the programmes must be generated by compiler, linker, and loader, or whatever, with a certain set of memory addresses built in, so that they are anchored in memory. Should the memory base address change (by a change of lowerlimit, or movement from one partition to another), programmes must be at least reloaded, and might need to be relinked and recompiled, depending on the sophistication of the software used. The methods are *dangerous*, particularly in a multiprogrammed machine, because there is nothing to prevent processes generating and using addresses in other processes' working spaces.

For those reasons, we shall concentrate on the function **get( )**. In single contiguous allocation, the function is executed exactly once for each process by the operating system as it sets up the process. The request for an area of memory of a certain size is answered either by a report of failure or by allocation of the required area and return of a segment descriptor (in practice, commonly just a pointer to the area's position).

What else can you do ? Within the constraints of single contiguous allocation, not much : you can either **resize** (change the size of) the area, or **release** it. The only essential restriction on such requests are those imposed by the physical limits of the available resources, but in practice it is quite hard to see how you could effectively use **resize**, and it is rarely provided.

To see why this is so, suppose you can indeed control the size of available memory through **resize**; how do you use it ? If you increase the size of your memory, it will stretch at the high address end. (If you think there's a reasonable alternative, work it out. Don't forget that paging doesn't affect the programme's belief that it is running in a flat memory.) How do you control the programme well enough to be able to make use of it ? We don't want to force everyone to use assembly languages ! Similarly, if you reduce the size, some space is chopped off the end : how do you know you won't lose something important ?

Well, perhaps in some cases a clever compiler will be able to make sensible use of the resizing function, and it's far from useless, but it doesn't solve all our memory problems. What does solve all our memory problems ? We know the answer from our earlier discussion, where we expressed it in terms of segments. We need more control

* - In the first version of this text, "years" was mistyped "tears". We changed it reluctantly.

over where the memory appears *within* the programme – though of course we must be able to manage this without disturbing the existing addresses. We would like to be able to request more memory to accommodate items which are *significant to the programme* – such as a procedure, or (more commonly) a data structure.

A fairly close approximation to this requirement is the common provision in languages such as C of functions with names like **alloc( )**. On execution, these return some sort of descriptor to an area of memory of a size provided as one of the arguments to the function. Whether or not this amounts to our **resize** function depends on how it's implemented, but there's usually no guarantee that the new memory will be addressable as a continuation of the old. On the other hand, it does satisfy our requirement for new data structures. It is probably best described as a new segment.

We can summarise our arguments by saying that there are good reasons why we might wish to use several segments in our programmes. It should be clear that to use a segmented memory is to move away from the flat memory model; but, in practical terms, the move is in perhaps the only direction which can readily be accommodated by commonplace processor technology with only minor extensions. In fact, instead of providing a single flat memory area, the segmented model gives you several flat memory areas – which we are already able to do with the base and limit registers ! The difference is that we shall want several areas within one programme. It is possible to manage the addressing with a single set of registers and a lot of juggling, but it is much easier if several sets which can be used simultaneously are available in the hardware, much as with the paging hardware, and this is the usual method of implementation.

ADDITIONAL FACILITIES FOR MULTIPLE SEGMENTS.

If we want to extend our memory management regime to cater for systems in which several segments of memory can be in use at once, we find a new set of problems. In fact, of course, we do want to use many segments, for at least two reasons : we expect that a process will naturally use several segments, and we want to run several processes at once. We shall therefore investigate further.

We can start by making the same sort of distinction as we did for the segments themselves; we can ask what determines the number of segments to be used, the system or the process(es) in question ? If the system determines how many segments we can have, the result is partitioning. While partitioning did a useful job in the early days of multiprogramming, it is far from being a satisfactory solution. Systems which use partitioning restrict the size of memory request which can be made, either offering only a single partition size, or at best a small range of fixed sizes. The origins of the areas are, of course, also predefined, typically at values closely related to 2^n; the constraint on memory size is a nuisance, but the fixed origin does mean that we can produce the code files before they're needed. Unfortunately, it seems that partitioning is about all we can do to support multiprogramming if we're restricted to a simple computer with only basic memory management hardware. We know that in many cases there will be a lot of memory that isn't being used, but it's scattered. We might find a little programme which could run in a spare area of memory at the end of another programme's partition – but if we try to do that, we have to relocate the new programme to a strange origin which can't be predefined, which rather spoils the point of the fixed origins, and which in any case we might find difficult because we (presumably) have nowhere to run the loader. Further, even if we're willing and somehow able to fit in the new programme, it might outlive the original occupant of the partition, and we'll be left with a truncated partition in which we might not be able to fit the next programme scheduled for that area.

For reasons of these sorts, partitioning is no longer common. It is much more useful to let the number of segments required be determined by what's happening in the system, which brings us back again to the **get( )** function. This must be used by the system itself when it sets up a new process, or by a process when it comes to a point where a new segment is required. If processes are expected to manage their own memory the **release( )** function will also be necessary, though it is reasonable to ask whether garbage collection, which is a universal requirement, should be handled by the operating system. The answer is clearly "yes", but few (no ?) systems take this seriously.

From the point of view of the process, the **get** function, perhaps with the associated **release**, is sufficient no matter what the environment – provided always that we can make the environment work. We've discussed this question briefly in our *HISTORY* review (*ONWARDS AND UPWARDS – OPERATING SYSTEMS*), where we noticed that clever implementations of partitions used special registers to do just what we want : *base registers* let us define a "partition" starting anywhere in memory, at any time; and *limit registers* gave us the means to check every memory reference to make sure it was within range. We would wish to change the partitioning system in three ways : first, both registers should be arbitrarily changeable to accommodate a segment of any size at any origin; second, we require a different pair of values for every segment; and, third, we require some sort of interlock to make sure that the areas of different segments can't overlap. What we commonly get is the first two, but without the limit registers. That means that there is no automatic guard against overlapping (which is just as well, because there are legitimate reasons for declaring segments completely enclosed within other segments – the real trouble is with partial overlapping), so any necessary safeguards must be provided by software. We might think of this as something approaching a rather unsafe implementation of segment descriptors, but at least it gives us hardware support for a sort of segmentation.

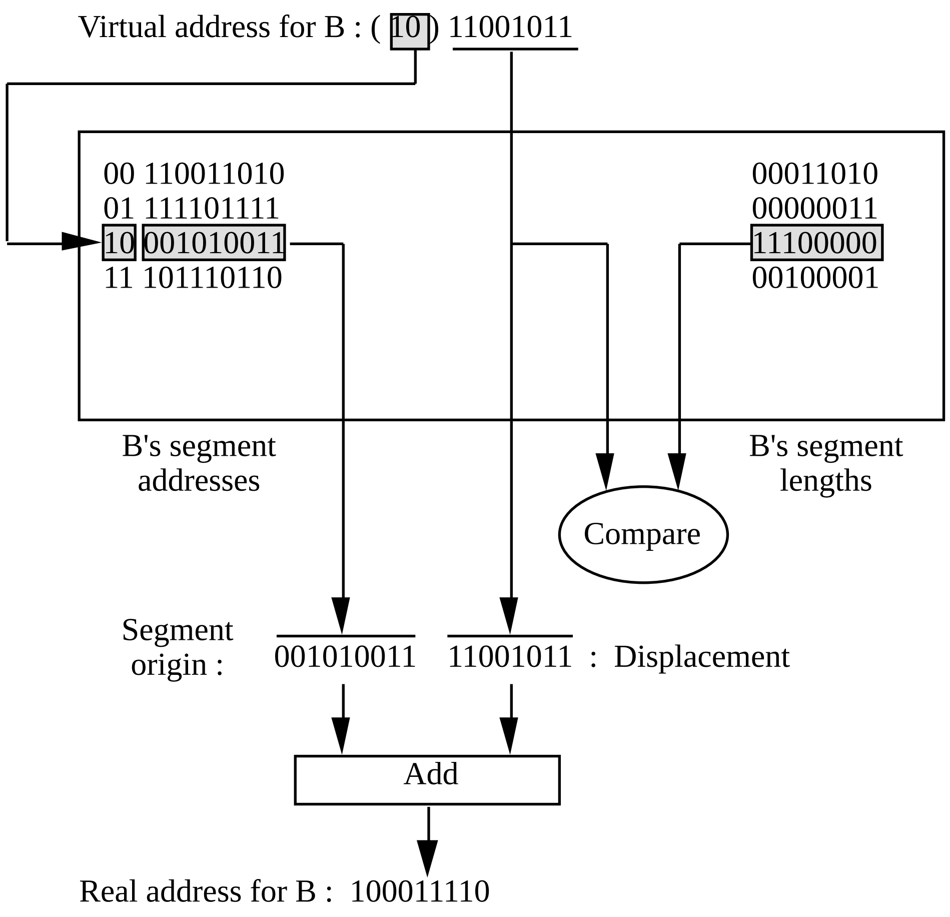
For a process which works with a flat memory model, these new facilities don't in themselves change anything, but the model is simplified in two ways. First, if we ask for an area of memory of *N* bytes, then we get precisely that; and limit registers, if they exist, ensure that there is no chance of accidental (or, for that matter, deliberate) forays outside the requested area. Second, the addresses within the area, so far as the process is concerned, begin at 0 and go up to *N* – 1. We can use the same code anywhere in memory without reloading, relinking, or recompiling. So long as we stay within one segment, we are, if anything, closer to the flat memory abstraction than we were in a simple system with a **lowerlimit** !

Of course, it's a little more complicated on the system side. We've added some hardware, and to make it work we have to manage the register values for each segment. Just what we do with them depends on the design of the system, but quite generally these register values must be specified in any full description of the process; we say that they are part of the *process state*. The operating system must look after the process state as part of its process management activities.

Generally, in a segmented system, the programme's view of memory encompasses several independent areas, each with an address range which begins at zero but with independent limits, and each with its own base and limit registers. As we expected from our discussion of the segmented memory model, the process will therefore have to give two-part addresses, identifying both the segment to be used, and the address within the segment.

Now the process no longer has to know before it starts just how much memory it will require. It can acquire more memory as it becomes necessary with the progress of the computation. Further, as the new memory comes in independent chunks, it can be allocated at any location in the real memory, not necessarily at the end of the existing chunk. The compiler for a programme running in segmented memory needs to be a little cleverer than one designed for a flat memory, but the added complexity is small and associated with the programme structure, about which the compiler knows all there is to know.

Here's a diagram of a possible segmented memory implementation. (It's tempting to call it *the* segmented memory implementation, as it's hard to think of anything substantially different for an underlying flat memory.) The addresses generated from the programme code are now no longer real memory addresses; they are called *virtual addresses*, each composed of a segment identifier and a displacement. The segment table is a collection of segment descriptors, as we described them earlier in *MEMORY MODELS*; it contains (at the left-hand side of the diagram) the full memory address of each segment's origin, as a segment can begin anywhere, and also (at the right-hand side) the segment's length. As the segment base address can be anywhere, the segment base and displacement (from the virtual address) must be added to find the physical flat memory address. The displacement is also compared with the segment length, and an invalid address exception is signalled if the displacement is out of range.

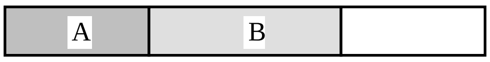


For convenience (and for easy comparison with the page table, which we describe shortly), we've shown the segment table as a traditional compact array, but that is by no means the only way to do it, and isn't even very convenient because it requires that you decide before you start how many segments you'll want. In practice, it is more convenient to keep the segment descriptors in memory just like any other variables. (It can be even more convenient to split up the descriptors, and to keep the segment lengths with the segment data; we shall explain why in *MEMORY MANAGEMENT : THE SYSTEM'S VIEW.*) The segment identifier in the virtual address then becomes the ordinary memory address of the descriptor, but the machinery is unchanged.

In practice, while segmentation can be made to work well, it is supposed to be comparatively expensive on hardware, so it is nowadays rarely found on its own. Whether this supposition is true with today's semiconductor fabrication techniques seems not to be thought worth worrying about. A number of paging systems provide segmentation as well as paging. This underlines the distinction between the two : paging, being dependent solely on the hardware, can in principle be used to implement any sort of memory management whatever, whether flat or segmented. In the segmented paged systems, the segments are significant for addressing and for memory protection mechanisms. We shall say more about this later.

PAGES.

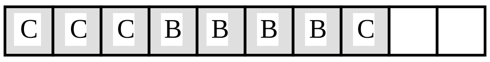
That gives us a means in principle to provide a segmented memory, but it hasn't solved all our problems – instead, it has made new ones. We might have removed the rigid constraints of fixed partitions, but all that has done is to return us (with somewhat more complicated hardware) to where we were before we had partitions. The operating system still has to look after the problems of fitting the several segments into memory, but now it can no longer rely on the benefits of fixed partitions. It's easy to illustrate the difficulty. Suppose we have three segments, A, B, and C, which occupy fractions 0.3, 0.4, and 0.4 of the total memory space respectively. Suppose, purely in the interests of describing what happens, that each is a single-segment programme. First A starts, then B starts : using the obvious memory management scheme, the memory map now looks like this :



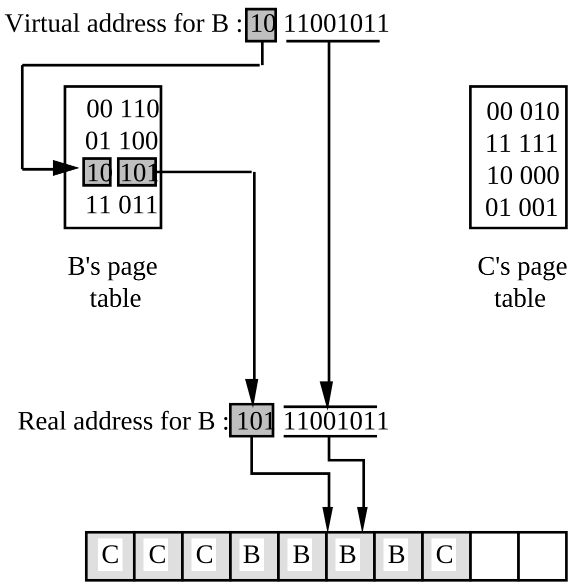
Now suppose that A finishes, and C attempts to start. C requires 0.4 of the memory, 0.6 of the memory is available, but there is no single chunk of size 0.4. This sort of problem turns up in one form or another whatever memory allocation strategy is used; it is called *fragmentation*.

If we decide not to move B, there is only one other way to solve the problem : we have to split C into (at least) two fragments. This split has nothing whatever to do with segments, which are semantically significant structures, and it must be managed so that the process's view of a single contiguous address space is preserved. To make that work, there must be a mechanism with which adjacent addresses within the segment can be converted into quite separate hardware addresses. We have the machinery to do that – some sort of base register – but to keep up the processor speed we would require several such registers, and we would also need a very fast way to pick the correct register for use at each memory reference.

There is such a way. It uses the memory address itself to identify the correct register; it's more expensive, but it is so effective that it is provided in most computers which allow multiprogramming. It works by splitting the memory into smaller chunks called *pages*, and packing the programme into them. Here is the problem solved using pages, each occupying 0.1 of the memory :



That solves the memory allocation problem – all we have to do now is make it work, and we can do that in very much the same way as we managed the segments. We must use a separate base register for each page, and find a means for selecting the correct register for each address. The register selection is easy if we make the pages 2^n words long, for then we can use the last *n* bits of each address as the displacement within the page, and the preceding bits as the register number, and it all happens automatically. Here's a picture of the two segments in a more haphazard arrangement, just to show that we don't even need to keep contiguous pages in the "right" order; the important point is that the programmes have no way of telling that they are dismembered.



This time we have a *page table* to link the *virtual addresses* generated by the code to the *real addresses* in the computer memory. In the diagram, the virtual and real addresses are 10 and 11 bits long respectively, and the pages are 256 words (or whatever addressing unit is used) in length. (Clearly, other values could be used for these numbers; this is an unrealistically small example.) Each page table entry contains a virtual page number and the corresponding real page number. Unlike the segment table, it doesn't have to carry the complete origin address because the displacement part of the virtual address is simply copied to the real address – and, because the hardware guarantees that the displacements must have the right number of bits, there's no possibility of a page overflow, so no check is necessary.

The important properties of the table are that the real page numbers should be writable, so that arbitrary page tables can be constructed to satisfy any arrangement of pages in memory, and that the translation from virtual to real page number should be very fast. The simplest organisation is as an **array [0 : 2^{m-n} – 1] of real page number**, with *m* denoting the number of bits in the virtual address. The dimension of this array is the maximum number of virtual pages, so the page numbers need not be explicitly stored (they are the array indices), but this is not convenient for very large memories; other sorts of associative memory can be used.

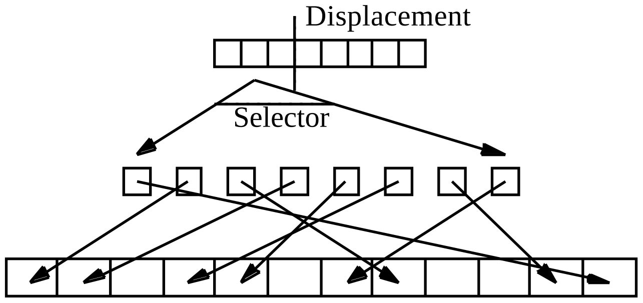
Just how we use this paging hardware is a matter for system design, and it comes down to how we make the virtual addresses. We've presented paging as a way of packing awkwardly sized segments into memory, but in practice it's usually used without explicit mention of segments. That doesn't mean that there are no segments, but it does mean that each programme is regarded as a single segment no matter what its internal structure. With this system design, each programme sees memory as a single contiguous block, and all the rest is managed by the system. Of course, all the advantages of segmented systems are lost, and software is back in the stone age, but this appears to be of little concern to the hardware designers.

An alternative is to use the segment displacements as the paging virtual addresses; with this architecture, we retain the segmented memory model, but use paging for packing the segments into memory. Now the segment identifier selects a segment length and a segment page table; the displacement is compared with the length, as in the simple segmented model, but also used as the virtual address for the page table to identify a required page. The hardware is a little more elaborate than for the separate systems, but there's nothing very complicated about it; those who worry about timing might like to observe that the main difference from a segmented system is the replacement of the address addition by a table index.

How are the pages allocated in the first place ? One way is to allocate all pages which might be required when the process is started, but as not all the pages are necessarily used that could be wasteful. An alternative is to allocate no pages in the beginning, but to acquire pages as they are needed. This requires an additional bit in the page table to mark whether or not the page has been allocated. These are initially set to show all pages not allocated, and are used in the obvious way.

PAGES AND OTHER THINGS.

Another way to look at pages is shown in the diagram below. This is equivalent to the previous diagram, but emphasises the use of the first *m-n* bits of the virtual address as a means of selecting one from a number of possible modules, which in this case are the real page numbers of memory pages :

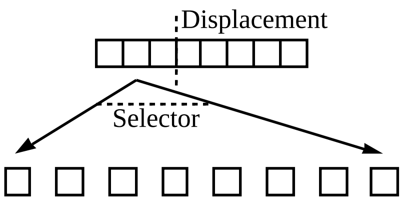


Splitting an address into two parts in this way is a convenient device for doing things with the relationship between computer and memory, so it's not surprising that its use isn't restricted to paging systems. Because it is implemented at the hardware level, the effects are invisible to the processor and can be very fast. It's interesting to compare the paging method with other related techniques.

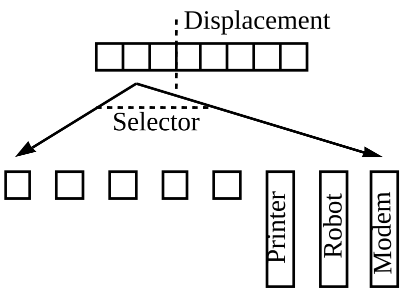
These are all based on the principle embodied in our earlier analogy between disc files and memory : that the processor doesn't really see the memory at all, but only input and output streams of data controlled in some way by instructions which it issues – again as streams – through its control and address bus connections. In "normal" operation, a processor will signal an address on its address bus, and give a read (or write) control signal, and something outside the processor will identify the corresponding memory address and return the bit pattern stored at that address on the data bus (or store whatever signal is on the data bus at the specified address). From the processor's point of view, though, it is quite immaterial what happened at the other end of the stream, and if something is returned on the data bus when it has issued a read instruction it is of no significance where it came from.

This leads to the general technique of *memory mapping*. We can attach anything we like to the processor buses, provided that it doesn't damage the processor, and use the control and address bus signals to drive it. The processor will see whatever it is as a part of memory. This is usually managed by using a few address bits (usually the high-order bits, though that isn't essential) to select the entity which responds, and the other address bits as some sort of control instruction.

Perhaps the simplest example is the multi-chip memory. If the processor address is *m* bits long, and the memory chips used can only accept *n* (< *m*) bits as their addresses, then several chips will be needed to build up the memory to the maximum size which can be handled by the processor. The leading *m*-*n* address bits must then be used to select the required chip.



If we now replace some of the memory chips by device-driving circuitry, we achieve memory-mapped input or output. Provided that the device-driving electronics responds to the processor's signals in just the same way as the memory chips, the processor can't tell the difference, and to the programmer it appears as though the devices have been wired into the memory in place of the memory chips.



MAKING IT WORK.

With either paged or segmented memory, the fine details of addressing are handled by hardware, as the processing would otherwise be very much slower. Both require tables – base register contents for paged systems, and base and limit register contents for segmented systems. These are called, reasonably enough, *page tables* or *segment tables* as appropriate; if we want to refer to whichever table happens to be there without being specific, we shall call them *addressing tables*. Like the base and limit registers from which they grew, these tables form part of the process state, and must be saved when the processor switches to another process and restored when it returns. There are advantages with both forms :

- It's easy (in principle, if not in practice) to move pieces of programme and data about.
- We don't need programme-sized contiguous segments of memory any more.
- It becomes easy to separate code from data : "pure code" now means something. In consequence –
  - it's easy to share code or data between processes;
  - recursion becomes trivial.
- Areas can be protected independently.

The notion of "pure code" is not very old. Not very long ago, it was quite common for programmes to overwrite their own code even at the level of individual instructions, typically changing the destinations of branch instructions, and for compilers (or, for that matter, assembly language programmers) to produce programmes in which code and data areas were mingled on a very fine scale. Hardware did the same : it was common for subroutine entry instructions to store the return address within the subroutine's code. (This was one of the reasons for recursion being seen as difficult; the data, and the return address, of a recursive procedure must not be shared, though the code is.) Practices of this sort led to the idea of *serially reusable* code, which could be used by any process provided that no two processes used it simultaneously.

A word in defence of the early programmers : They weren't stupid; they were just working in a different world. How would you go about copying an array from one place to another in a machine which had no index register, and possibly no indirect addressing ? And if you only had 4K of memory available, you would keep data as close as possible to the instructions which used them in order to make best use of the limited range of relative addressing, which saved memory. A quotation[EXE16] *:*

"A scrutiny of these instructions at once reveals a difficulty. The addresses specified in the two first instructions depend on the value of r, and so change from one cycle to the next. Clearly this will not do. Some provision must be made in the program for increasing both of these addresses by unity during each cycle. This can be done quite simply by exploiting the fact that instructions are coded as sets of digits, just as numbers are. Instruction-words, just like number-words, can thus be altered by performing arithmetic operations on them. Clearly what is needed in the present case is to arrange matters so that the patterns of digits which represent the two "awkward" instructions are modified arithmetically each time they are obeyed. ..."

These implementation problems undoubtedly contributed to the considerable confusion which was evident in the diverse interpretations of the notions of process and programme. Until the possibility of using one code segment as the programme for several processes became commonplace, there was no practical illustration of the distinction.

SHARING MEMORY BETWEEN PROCESSES.

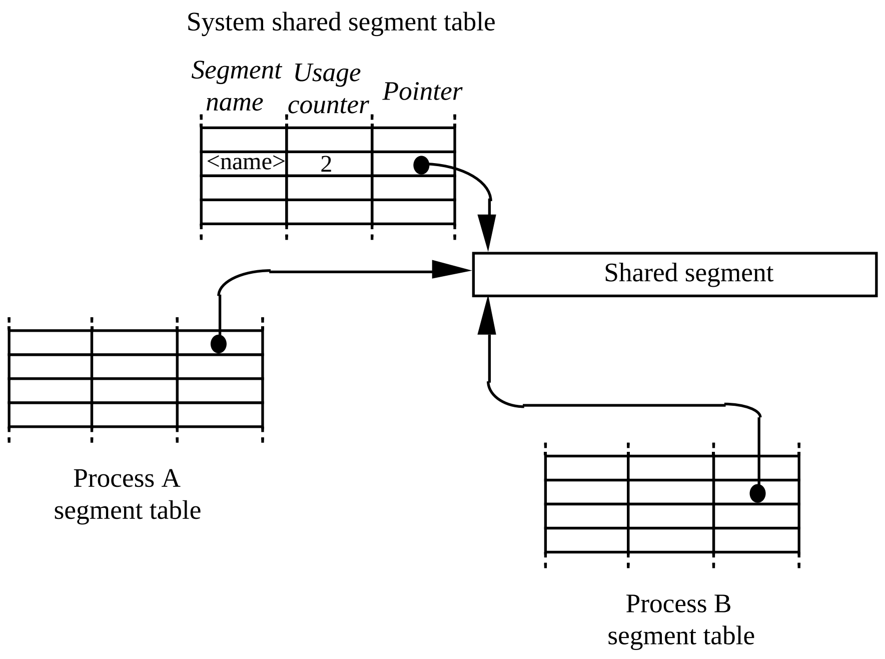
In the chapter *MEMORY MODELS* we suggested that it would be useful for processes to share memory, and now we take up this topic again. We have not done so before because without the memory management techniques we have been discussing, sharing memory is at best awkward, and at worst dangerous. Methods depending on shared base pages or arbitrarily allocated memory areas (such as the **common** area we mentioned earlier) which had to be addressed directly were not very satisfactory, but the segmentation and paging techniques give us a new way to achieve the same end, and it's much more orderly.

What happens when memory is shared ? The result is that the same area of the computer's memory must be included in the address spaces of two or more separate processes. In view of the considerable effort which has been put into operating systems to keep address spaces separate, this is an odd thing to do – and, because that effort has given us structures like base and limit registers which restrict access to memory very effectively, it is also rather tricky to do. It is easy enough to manage shared memory when *all* of memory is shared, and indeed that is how monitor system routines were made available in early multiprogramming systems, but in a system where memory is secure that avenue is no longer open to us. Our customary technique for allowing programmes access to areas outside their own disjoint memory spaces has been to provide system calls; we can certainly do so, but at some cost in administrative overhead, and one of the advantages of the shared memory is its very high communication speed.

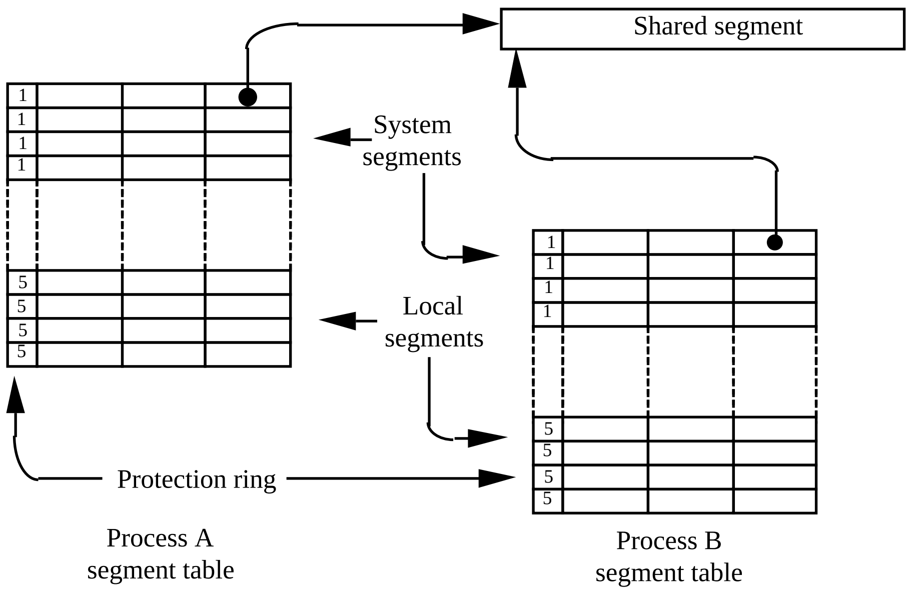
The basic idea is simple enough : if we want two processes both to have access to the same chunk of memory, then all we have to do is to put a pointer to the same chunk in the addressing tables of both processes, and, hey presto, there it is. The simplicity of the basic idea doesn't mean that the implementation is trivial; to make this work properly, the memory management system must be quite clever. To establish the shared memory, the several processes concerned must all make their requirements known to the system, and it is no longer possible to attribute the memory to any single process. It is usually expected that the shared memory must conform to the memory model assumed in the programming language used. The "sharedness" appears to each process concerned simply as an attribute of what is otherwise an ordinary piece of memory.

This leads us at last to another piece of the API, which we foreshadowed in *MEMORY MODELS* – the function **share( )**. Now we must define it more carefully. We begin by deciding what sorts of parameter will be necessary. We observe that to a single process it looks much like a decorated form of **get( )**, so **share( )** should reasonably begin with the same parameters as **get( )**. That deals with the memory model. Now we must add whatever is needed to cope with the sharing. That must certainly include matters of security : obvious items are what sort of access is required to the shared area, and some control over which processes are acceptable as sharing partners. It is up to the operating system to provide whatever special administration might be required. Our discussion suggests that two such provisions are clearly necessary.

- *Access* to the shared memory is the first requirement. The system must allow the same area of memory to appear within the visible memory of two or more different processes. From the language end, this is comparatively straightforward if we are using a structured memory model, for then the shared area can be treated as one of the structural units, and implemented as a segment. Gaining access to the structure is not so easy : as neither process can tell what the other is doing, how can both make links to the same area ? A straightforward solution to this problem is to let the operating system look after the linking. In this method, each process requests (using a system call) access to a shared segment with some agreed name, identifying other processes which may share the segment. To support this method, the system must maintain its own table of shared segments in which is recorded the name and permitted processes. On receiving a request, the system searches its table; if the name is not found, then this is the first request, so the system reserves the segment and makes an entry in the table recording the segment details and the permitted processes; otherwise, this is not the first request, so the process identity is checked against the list and access permitted if appropriate. The diagram illustrates the structure; we shall discuss the *usage counter* shortly.



We saw previously that early systems sometimes simply identified a piece of memory which was reserved for use as a shared area. With virtual memory, a similar approach is to make some area of the addressing table standard for all processes, so that certain virtual addresses always point to specific points in the operating system. This is illustrated in the diagram :



- *Protection* is very obviously the second requirement. (We would make it the first if it were not that we must implement multiple access before there's anything to protect.) The possibility of protection must depend on whether or not the operating system can recognise the items to be protected. A base page which is merely a small area of memory with special behaviour is hard to protect; other implementations, depending on segmented memory supported by the operating system, lend themselves to protection much more readily. Another technique is illustrated in the diagram above; here the system uses the protection ring technique to ensure that the process makes only legitimate use of its address table entries.

Once access can be effected, the system must be able to ensure that the shared areas are used in an orderly fashion. For read-only access, as in the corresponding case of read-only files, there is no difficulty, as processes cannot interfere with each other. For reading and writing, we almost always need some means of coordinating the access of different processes, or two processes might try to write into the same area at the same time, leading to unpredictable results. We discuss such techniques in the *PROBLEMS OF CONCURRENT PROCESSING* chapter.

If memory is shared, how is the system to know when it is no longer required ? The shared area can only be repossessed by the system when it is no longer in use by *any* process, so we must find some means to determine when all processes which use the area have finished with it. This is usually managed by maintaining a *usage count*. When a process begins to use the area, the operating system increases the usage count by 1; correspondingly, whenever a process finishes with the area, the system decreases the usage count by 1. The first diagram above illustrates how this can be managed. The system can then retrieve any memory area with a usage count of zero. This works well – provided that all processes follow the rules, and none breaks down while using the shared memory. If that happens, the usage count might not be counted down correctly, and memory will appear to be in use when it could in fact be released. This is a cause of *memory leakage*, which we shall discuss later.

A special requirement for sharing memory areas is that the parties involved must agree on the structure of the areas. The earliest example of shared segments (apart from accidents of implementation exploited by assembly language programmers) was perhaps Fortran's **common** area, which we described earlier. The **common** area was just that – a raw segment of memory, with no structure whatever. It was declared in each programme unit in which it was used, and the declarations could be quite different. There was (by definition !) no checking, so even a change in the order of declarations in the **common** area could effectively destroy a programme. This is not a very good way to manage the sharing. Later, a more disciplined approach to shared areas evolved, based on facilities to share source code (typically declarations) between different programmes. In general, though, without information about the structure of the areas which is only accessible to the programmes, the operating system can do little to provide protection against such abuse. We conclude that such uncontrolled sharing is not a good way to implement shared memory; at the least, it should be organised in such a way that it is certain that the processes involved in sharing agree on the identity of the things shared.

Attempts to share memory areas between programmes written in different languages are even harder to manage. The only successful general implementation of such a system of which we are aware is Linda. We have already mentioned the Linda memory model; the shared areas (the tuples, which can be of any size required) are implemented in a separately managed memory space, and a set of procedures for access to the memory are provided in each language in which access to the shared system is required. The trick is worked by not using sharing supported by the operating system at all ! (Of course, the complications don't disappear – they are transferred to the area of *interprocess communication*, which we shall discuss later in this section.)

PROTECTION AND SECURITY.

Having invented addressing tables, we find we have somewhere to store information about the areas of memory we allocate. An important part of this information is to do with safety. Clearly enough, this is very important if we are sharing areas of memory between different processes, when untoward interference between them must be avoided. Here are three examples.

Bounds checking : If we know the size of every area, we can check that there is no attempt to address any part of memory outside the area. Just how effective this is depends on the unit of allocation, and the machinery available to do the job. With a paged system, where the memory allocation has little to do with the programme structure, all we can do is check that addresses generated by the code do actually exist. With a segmented system, we can do better, because now we can, for example, allocate each array as a separate segment, and detect every attempt to address nonexistent array elements. This is really only effective if implemented in hardware; you can do it in software, but it's fairly expensive – also, it's something the operating system can't do much about.

Protection rings : We've already seen how protection rings can be used to control access to operating system addresses. Each segment, code or data, is given a protection ring number. Every process also runs in a protection ring; the process's ring number is checked against the ring number of every data area which it attempts to use. Access to less sensitive areas is permitted without check; but attempts to use more sensitive data result in interrupts handled by the security system.

Not only data can be protected in this way. Procedures can also be allocated at high protection levels, and access to these controlled in much the same way as with data.

Capabilities : Protection rings work well when the resources to be protected can be arranged in sets, each of which includes all sets of higher sensitivity, and in which the same security classifications apply under all circumstances. Typically, that makes it useful for controlling access to the operating system, which behaves in the same way towards everyone who uses the computer facilities, but it's not necessarily as useful for protecting general software and data which might have much less tidy requirements. Capability systems can be used in these circumstances : as with capability systems in general, each segment in memory, code or data, may be associated with a capability, which must be presented by the subject if access is to be granted. A significant advantage of a capability method is that it can be made to work in a distributed system, making it possible for processes running in separate computers to share memory in an orderly way.

Our discussion so far has been concerned with permitting or denying access to the segment of data or code, but that doesn't exhaust the protective measures we would like for safe use of shared memory. As we saw earlier, there are also requirements for internal consistency – the different programmes using the area must agree on how it is to be used.

From the operating system's point of view, the simplest way to address such questions is to ignore them. The system provides facilities with which different processes can set up shared memory areas, but what happens after that is left to the programmers and the language support software (compilers, etc.) to sort out. This is quite a common technique. It can clearly be abused, but without access to the internal structure of the areas there is little that the operating system can do about it. The most that can be done is perhaps to make it easy to share segments of memory, so that programmers can use the system facilities to share the logical memory structures – even down to individual variables – individually, rather than collecting them into larger clusters in which distinct logical entities cannot be distinguished by the system. Perhaps object-based systems provide facilities which approach this requirement. If the system doesn't even have distinguishable segments, then there's very little possibility of achieving any protection measure which requires that information about areas of memory be kept by the system, except by introducing special structures in particular case, which is a nuisance.

It might be that this laissez-faire attitude is appropriate to shared areas about which the operating system knows nothing, but it is less defensible when adopted as a policy for shared areas defined by the operating system. "Base-page" areas, and similarly defined repositories of system information, are a case in point. The early Macintosh systems suffered from interference between different programmes caused by indisciplined use of a base page. The most satisfactory answer to this problem is undoubtedly to get rid of the base page. Any required access to system parameters can be better supported through supervisor calls, which make it possible to provide appropriate protection to each parameter individually; for communication between programmes, ordinary shared segments should be used.

COMPARE :

Lane and Mooney[INT3] : Chapter 10; Silberschatz and Galvin[INT4] : Chapter 8.

REFERENCE.

EXE16 : S.H. Hollingdale, G.C. Tootill : *Electronic computers* (Penguin, 1965), page 131.

QUESTIONS.

In this chapter, we've relied very heavily on the idea that a programme will keep running provided that its code and data can be got into memory when required. Computists have done that for themselves with data for a very long time – they keep data in files, read it into memory as required, use it however they want, and write it back after use. (A database system is an extreme example.) Why don't they do the same sort of thing with code ? It's more obvious than overlays. Or is it ?

What is the simplest way of implementing the memory management functions mentioned in the *MEMORY MODELS* chapter ? Consider in particular the sort of information which must be generated and manipulated. What should happen if a function cannot do its job ?

Can distributed systems be used as ways of providing appropriate hardware for operations requiring different memory models ?

What must the system do to make sure that it can properly administer shared memory in case one of the sharing processes breaks down while using the shared area without decreasing the usage count ? If a process can make several simultaneous calls on the same shared area (perhaps by executing procedure recursively) does it make any difference ?

If the charge for computer use includes any component for use of memory, who should pay for shared memory ? How can the accounts be kept ?

Consider what a "dangling pointer" does to a memory management system. What must a system do to guarantee that no one but the owner of the pointer can be hurt ?

Why don't paged systems require limit registers, while segmented systems do ?

Suppose that you are implementing a compiler for a language which includes some sort of **get** operation. (**malloc( )** in C, **new( )** in Pascal, etc.) The compiled code is to run in a flat memory. Where do the newly allocated items appear in the process's memory space ? What happens to address checking ?

Why might it be useful to allocate segments within segments ? (Consider writing a memory manager for a machine with pure segmented memory, and think of other examples.)