VIRTUAL MEMORY

WHAT IS VIRTUAL MEMORY FOR ?

There is strong anecdotal evidence for the existence of some law of nature which guarantees that whatever is currently deemed to be a useful programme will require rather more memory than is in your computer. The only known exceptions to this law are observed in multiprogramming systems, but here the law applies in modified form : the total memory requirement of all the programmes running at any moment is rather greater than the memory available.

That's a nuisance, but if our programme is too big we can grin and bear it if we have to. More seriously, the same constraints lead to interference between programmes which can destroy our functional system. This takes the form of a phenomenon called *deadlock*. If each of two programmes running concurrently owns 0.4 of the system memory, and then both request an additional 0.3 of the total, what can the system do ? We deal with deadlock in more detail later, but for the moment observe that it is a serious problem in any system in which a finite resource is allocated.

We have a sort of solution for single-programme systems, and we have already met the overlay methods which were used to overcome the phenomenon, but these can hardly be seen as a satisfactory answer, as they require a lot of careful planning by the programmer, which doesn't fit in with our aims. Further, they don't extend at all to multiprogramming systems. Therefore, we assert that it is the operating system's job to provide whatever support is needed to make it easy to get work done on a computer, even if the support needed is a bigger memory. It is therefore a fortunate and happy fact that we can use our new memory management skills, not to mention hardware, to construct a new and completely automatic solution, which we call *virtual memory*. In principle, this is a simple extension of the addressing techniques which make it possible for a programme to be scattered piecemeal throughout memory. We merely allow the scattering to extend over both internal memory and disc. In practice, we have to do a few additional tricks in the hardware and software to make it work effectively, but the idea is just as we have stated.

We also add a new word to our vocabulary : *swapping*. We use it to describe the virtual memory system's movement of information to and fro between memory and disc. We speak of "swapping in" and "swapping out" when discussing the copying operations, and say that an area of a process's memory is "swapped out" if there is currently no copy of the area in memory, but a copy is available on the disc. We emphasise that swapping is an activity of the operating system only; it is quite imperceptible to the process, which continues to enjoy whatever memory model it thought it was using. The ideal of a functional system is therefore preserved.

It is also tested a bit, not in itself but as an adequate definition of the sort of system we want. We assumed that the aim of the system could reasonably be expressed as the satisfactory execution of functions, which by definition are independent of time. That was certainly true for a batch system, and in practice worked well for interactive systems; a slow-running system would make people cross, but wouldn't affect the performance of the programmes. This is no longer so obvious. As people want to use computers for activities such as real-time animation, a slow-running system can be a catastrophe. We shall address a few of the issues later, but perhaps this observation suggests that we should take time into account right from the start. As with security, if you don't build it in from the beginning, it's hard to bolt on later.

Before leaving this introduction, it is worth emphasising that the virtual memory which we are discussing is *primary memory*. We saw at the beginning of the *WHY MEMORY ?* chapter that the major reason for having primary memory is that it's fast; the other reason (which follows from the first) is that the processor is designed to work to and from primary memory, so we have to get code and data into memory before we use them. Typically, though, the code and data come from a disc, so, if the code contains few loops and branches, the material might merely be copied into memory, used once, and then discarded. If that's so, then the speed of the memory is ineffective, as the speed is likely to be governed by the rate of transfer from the disc. In such circumstances, we're really only using primary memory for the second reason, which is an accident of design. The advantage of having memory is only realised if we want to use an item in memory *repeatedly*.

We make this comment to point out that virtual memory is often used inappropriately. Suppose that we wish to make a few changes to a large file. By the wonders of virtual memory, we have an addressing space of 64 megabytes, so we copy the file into memory. Of course, most of it is virtual memory, so most of the file is copied into memory by the programme which reads the file, and then, as the copying continues, swapped out to the disc by the virtual memory manager. We execute the programme, and make changes to perhaps a few hundred positions in the file, for each of which a sector must be read from the disc, changed, and put back again. Then we rewrite the file from "memory" to disc, by fetching all the data from disc into primary memory, and writing it to disc again.

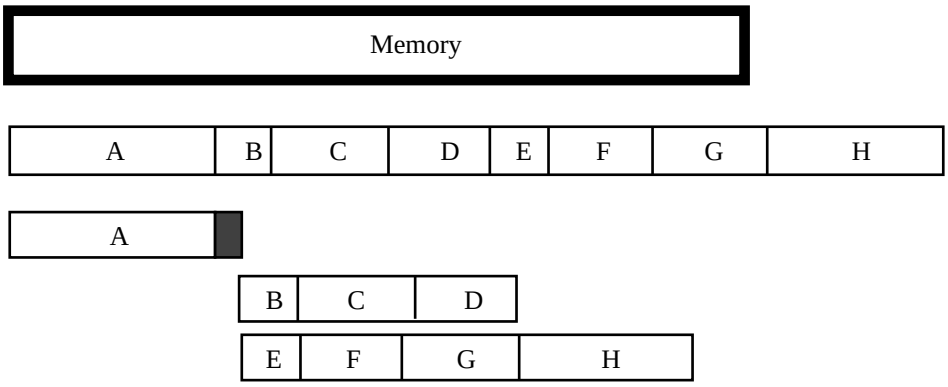
We have (or, rather, the computer has) done a lot of fairly unproductive work. We (now the programmers) haven't necessarily done it frivolously; we have written the programme as simply, quickly, and reliably as possible, and relied on the operating system's designer's claims that the system is there to do things efficiently for us. What went wrong ?

The problem is one which we have already met (in the chapter *WHY MEMORY ?*) : we have two quite different ways of providing a very large memory – virtual memory and the file system – and in the example they clash. The solution, at present, is for the programmers to take more note of the system constraints and organisation, and to write their programmes taking account of the fact of the file structure of the data. This intrusion of accidents of implementation into programming is unfortunate, but inevitable with current practice. The problem might eventually be solved by the very large address spaces we mentioned earlier, which in effect mean that we can leave everything in memory (virtual, of course) permanently; but until then the integration of secondary memory, virtual memory, and primary memory is likely to be less than perfectly satisfactory.

OVERLAYS AGAIN, AND LOCALITY OF REFERENCE.

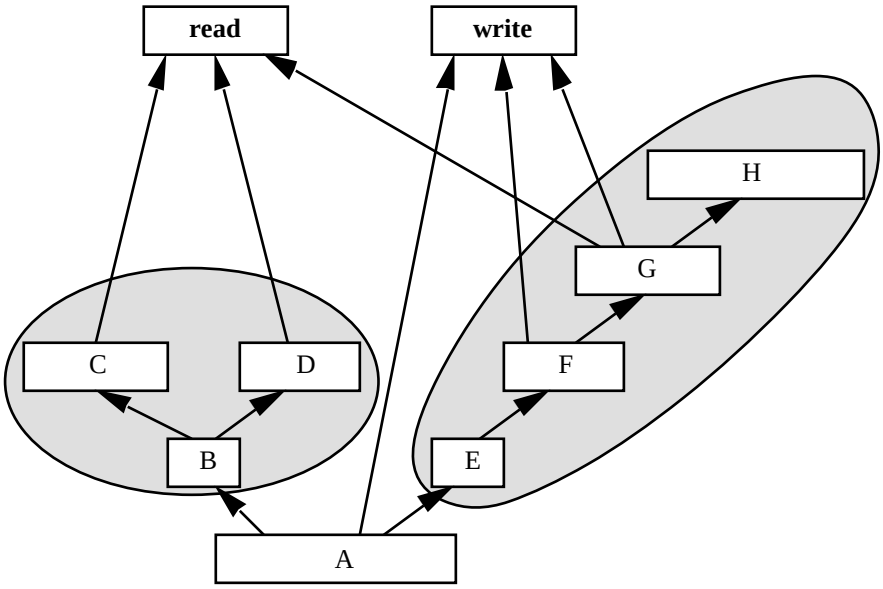
If the programme is too large to fit into the available memory, you might still be able to run it using *overlaying*. We described this technique earlier, in the chapter *MEMORY TRICKS*; it requires no additional hardware resources, and can be considered as the private concern of the process, the linker, and the loader. We'll describe it in some detail because, though so far as we know it isn't used seriously any more, it introduces in a rather clear way a number of ideas which we'll need later.

The principle behind the overlay technique is that a programme can run if the code and data areas which it is using at the moment are in memory, and provided that we can maintain this condition it doesn't matter what happens to the rest of the code and data – and, in particular, it could just as well be saved on the disc. We can therefore run programmes which are too big for memory by providing means to exchange the contents of memory with a copy on the disc in such a way that the active parts are always in memory. The diagram below shows how this is managed in an overlay system. We use the programme example we introduced earlier, and a carefully contrived memory which is rather too small to accommodate the whole programme at once.



A is the main programme, and remains in memory throughout. The rest of the programme is split into two sections, { B, C, D } and { E, F, G, H } – notice that these follow the static programme structure shown in the earlier diagrams, so that there are no procedure calls from one section to the other. Each of these *overlay blocks* will fit into memory with A, even though the whole programme is too big. The short grey tail on the A segment is the administrative software, which intercepts calls to any procedure in the overlay areas from outside and ensures that the correct overlay block is loaded into memory before executing the real procedure entry. (Calls within an overlay block to other procedures in the same block or to procedures permanently resident in memory need no special treatment.)

In practice, while a tree structure is often a reasonably good approximation to the static structure of a programme, it must often be extended a little to give a realistic representation of memory requirements. That is because there are often many service procedures (input, output, etc.), either within the programme or provided by the operating system, which are widely used throughout the programme. The structure of our example might therefore really be closer to a lattice than a tree, like this :



Fortunately, that fits into the overlay method very well, with the main programme and the service procedures permanently in memory, and the middle parts handled as overlay blocks.

This structure is set up as a cooperative effort between compiler, linker, and loader, and, given the structure, the details are fairly obvious. It is much more interesting to consider why the method works, and an instructive approach is to begin by inspecting a model of programme execution which doesn't work. To work out what happens in the overlay system, we must have some idea of the patterns of memory use as it operates. Perhaps it would not be unreasonable to guess that a passable approximation to the memory access requirements of the programme would be a random distribution of references throughout the whole programme chunk, continuing for as long as the "programme" was supposed to be in "execution".

We can easily apply this model to our overlay example. The areas of the three programme blocks are in the approximate ratios

area of A : area of { B, C, D } : area of { E, F, G, H } :: 45 : 60 : 100.

Using the random access assumption, the probabilities of references to the three blocks are about 0.22, 0.29, and 0.49. The probabilities that the two blocks are in memory will be in the same proportion as the probabilities of reference to the blocks, as at any time the last block referenced will necessarily be in memory :

probability that { B, C, D } in memory : probability that { E, F, G, H } in memory :: 60 : 100,

so the probabilities themselves are about 0.375 and 0.625 respectively. The probability of a reference to the block not in memory is therefore

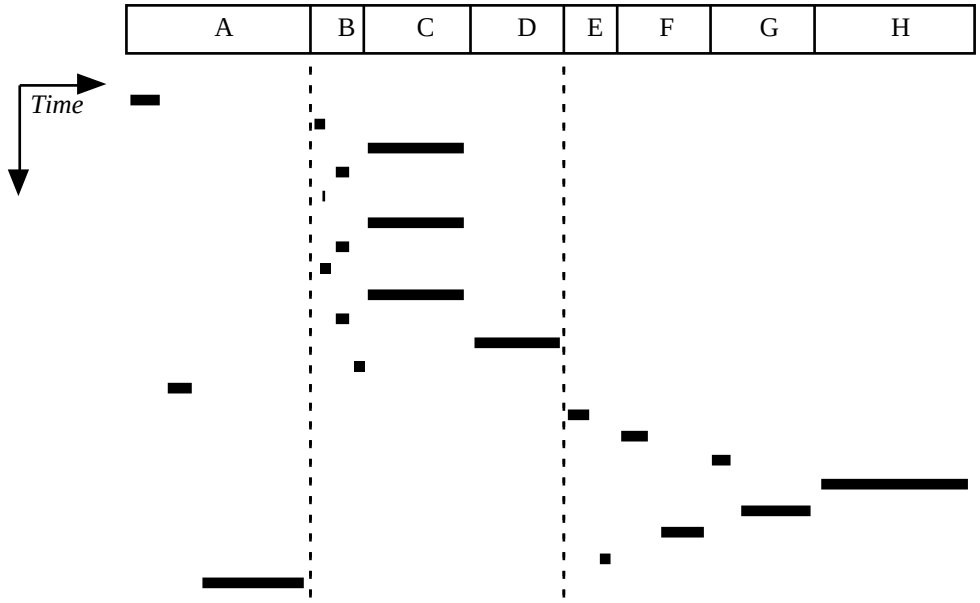
0.29 x (1 – 0.375) + 0.49 x (1 – 0.625) = 0.36, approximately –

which is to say that something like one in every three references to memory will require that a new block be loaded from the disc. And, yes, we know that this estimate is very crude indeed, but the order of magnitude is all we need, and the point is that the assumption of random memory references leads to this conclusion.

The point is also that this conclusion is quite wrong. Recall our comment in *WHY MEMORY ?* – "The file stores we usually use now are far too slow to be used directly as working storage when executing programmes, so we have to invent short-term storage, which we shall call *memory*, which we can use for this purpose.". If the random assumption were true, then executing the programme using the overlay method would (in this example) require a file store reference approximately once in every three instructions executed, and that in turn would give us a speed which is only a factor of three or so better than that we could achieve without any high speed memory ! In practice, we know that doesn't happen – so the random reference assumption must be very poor indeed.

Fortunately for the overlay methods, and for other important methods we shall discuss later, the random reference assumption is indeed very bad. That's because we don't write random code, and we don't scatter our data randomly through memory. Instead, we use various code and data structures which we have found by experience to be effective in composing good programmes, and which by their very natures collect together related material into localised areas of memory. The aim of any computer programme is to convert some collection of data items into some defined set of results, but the incoming and outgoing items are usually represented by fairly compact data structures. Similarly, we group the code for a particular operation into one or a few well defined procedures, which can be used by other well defined and compact procedures as required. Over any short period, then, the memory references required by a programme are likely to be restricted to a few quite localised areas of code and data.

Here's a possible memory reference pattern for our example programme. To draw the diagram, we've had to assumed a specific programme, and we've used an unrealistically simple example. With a more realistic programme, the density of use within the various sections would be increased, but the division into sections would remain – and could even become more pronounced if the more detailed programme structure corresponded to more persistent activities.



This phenomenon is called *locality of reference*. It is clear that the assumption of randomly distributed memory reference is quite wrong, and, because of the localisation which occurs in practice, the overlay method is effective. Most programmes settle down to operate smoothly with only a comparatively small demand for real memory : over any short interval, they are only using a small proportion of their code and data, not making memory references randomly over their whole address spaces. This phenomenon is called *locality of reference*, and the set of areas currently in use is known as the programme's *working set*. The size of the working set is determined by the programme structure, and the way in which data are used. It is probably rather obvious that we've worked out the diagram by hand and guesswork, but it's possible to measure them experimentally by recording the signals on the processor address bus and plotting them against time. Examples from several programmes have been measured[EXE23], sometimes with recording times covering 10^9 memory references, and shown together with computed working set size. The results are interesting, and in accord with our discussion here.

Though our example doesn't demonstrate it, it is found experimentally that locality of reference works for both code and data, and you can follow through a similar argument for the data based on the pattern of data use in a programme : typically, transformations are carried out by a succession of operations between structures P and Q, then structures Q and R, and so on, and only the structures currently in use must be held in memory. The arguments for data are not quite as good as those for code, because adjacency is much less significant for data than for code. Adjacent code words are very likely to be used at about the same time, because of the machinery of the computer hardware, but adjacent data words are not so constrained. A classic problem data structure is the two-dimensional array, which is sometimes used row by row, and sometimes column by column; there is no way to store it which will preserve a small working set in both cases.

What are the neighbourhoods within which memory references are likely to be collected ? They are areas of code or data which are related in the programme by being concerned with some common task. They are, in other words, the segments we defined earlier. The diagram above demonstrates this, given goodwill and a little imagination, but it's much clearer if you look at a real memory trace. We'd therefore expect the working set to be a collection of related segments of the programme.

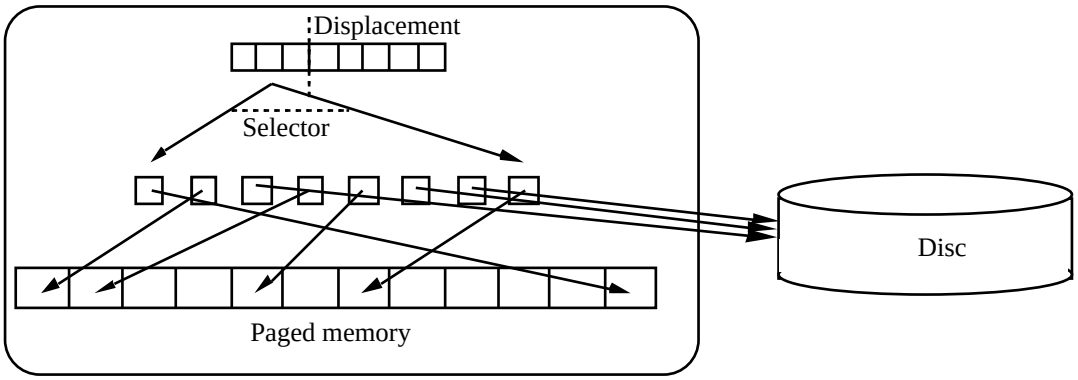
Of course, locality of reference didn't just happen when overlay systems came along. So far as the code is concerned, it's an immediate consequence of the processor architecture, in which the instruction address is automatically advanced by 1 after each instruction, and the common use of loops in programmes; and we usually keep related data close together because it would be stupid to do otherwise. These natural patterns were formalised and accentuated in structured programming techniques, where an essential notion is that you use different pieces of code to perform different tasks; it follows immediately that while you are performing a particular task, only the code for that task is active.

We've assumed throughout that we have a truly random-access memory. Just to illustrate the importance of that assumption, consider the IBM 650. This was one of the very early commercially available machines, and it was designed before anything now recognisable as fast memory was available at affordable prices. Instead, it had a set of forty electronic registers, with its main memory provided by a rotating magnetic drum. Because of this architecture, only a limited portion of the main memory – that just about to come under the read heads (one per memory track on the magnetic surface) – was available at any moment. In this case, the optimum layout for a programme was such that the next instruction for execution was always just about to become available from the drum, which meant that instructions had to be scattered about the memory. This was managed automatically by SOAP – a Symbolic Optimising Assembly Programme.

HOW TO DO IT.

The next task is to use our knowledge of locality of reference to develop an automatic method for getting the working set into memory, rather than relying on the cumbersome manual organisation of overlays. This turns out to be almost embarrassingly easy. The principle of locality of reference implies that if we refer to some memory address, then it's pretty likely that we'll want to refer to others in the close vicinity quite often in the near future, and we related this to the programme's segment structure above. We would therefore expect good performance from a policy which would bring into memory any segment referred to during the execution, because that would automatically load addresses which would be required in the near future.

Consider, then, what must be done to make the virtual memory work as we want it to. As our programme is executed, it continually generates memory addresses for both instructions and data. Each of these is presented to the memory management hardware. The hardware resolves the virtual address as we described earlier, and identifies the correct memory area's details. Under normal circumstances, it now expects to find the actual address of the memory area in real memory, and to proceed with the address calculation as we have already discussed. With virtual memory, it might be that some of the addresses turn out to be in areas which are swapped out. In a paged system, we can see this as a straightforward extension of the memory mapping ideas we introduced earlier. Consider this modified version of the paged memory diagram we presented :



Now certain selectors identify areas on the disc, not in primary memory.

But in that case the system can no longer function as we specified in the earlier discussion : it cannot respond as though it were fast memory, because it is physically impossible to retrieve the required data from the disc at sufficient speed. (If it were, we

wouldn't need fast memory.) It is therefore necessary to suspend execution of the process – and this leads to the idea of the page fault, which we shall shortly describe. Essentially analogous ideas apply to segments, though the neat "memory mapping" analogy doesn't work quite as well.

In order to make all this work, we therefore require that there should be additional information in the process's address table :

- The *present bit* : means "this area (page or segment) is present in memory";
- The *disc address* : where the area will be found on the disc if it isn't present. (The address is allocated either when the programme first defines the memory area, or at the first demand for swapping out.) Some systems keep either the disc address or the memory address; this saves memory, but you have to allocate a new vacant disc address if the area is swapped out (very fast for pages), and you can't save time on unaltered areas as described below.

If the area isn't present, then the system has to make it present. If there's a copy on the disc, it must be fetched; if this is the first reference to a data area, there might be no existing copy, so the system need only find space in memory. The system then proceeds with an appropriate set of these operations :

- Stop the current process until the copy is done (so the user isn't charged for the overhead);
- Find a place in memory for the area;
- Find its position on the disc (from the address table);
- Request the copy from the disc managing software;
- When the area is copied, change the address tables;
- Restart the process.

That's a *page fault* or *segment fault*. We shall use the term *addressing fault* if we wish to avoid committing ourselves. There is obviously a lot of work involved, and indeed an addressing fault at best causes a significant interruption in the smooth running of the system. While there are clear benefits to be gained from virtual memory, it is in our interests to reduce the number of addressing faults to the minimum consistent with getting the necessary work done in an acceptable manner.

FINDING SPACE IN MEMORY.

This is the second item on the list of actions above, and it might be by far the most complicated. It's only easy in the simplest case, which is when there's already a free area of memory which can be used to satisfy the request.

With a segmented memory, even that isn't straightforward, for we have to select which available segment to use. (In a paged system, all pages are the same size, so any vacant page will do.) This is the *allocation strategy*, which we discussed earlier, in *MEMORY MANAGEMENT – THE SYSTEM'S VIEW*.

Swapping is the next step. The system must choose a victim page or segment to banish to the disc; to do so, it uses a *replacement strategy*. The ideal replacement strategy is to swap out the area for which the time until the next demand is the maximum; unfortunately, there is no way to determine the time until the next demand, so we have to guess. There are several criteria which we can use for guessing :

- *size* – important only in a segmented system. It is difficult to devise a fair strategy which takes account of size; the obvious "swap out the first segment you find which is at least as big as the space required" is unduly hard on any big segments, which are at risk of being banished to disc on almost all requests, but to restrict swapping to segments of close to the required size leaves very large or very small segments almost undisturbed.

- *usefulness* – ideal, but unattainable. There are several ways to guess the use*less*ness of areas, some of which are :

- **Cyclic replacement :** This strategy is extremely easy to implement. We assume that simply working through memory page by page, or segment by segment, will be likely to pick old areas for replacement. If this were the only thing that happened in memory, it would work some of the time, but it isn't, and doesn't. Its main virtue (apart from easy implementation) is a sort of fairness : being quite unselective, it has no bias.

- **Least Recently Used :** While there is still no guarantee that the area which has not been used for the longest time won't be needed in the next few microseconds, the argument works the other way round : if an area will never be used again, then the time since its last use will go on increasing, so under ordinary circumstances (which don't include thrashing – see the chapter on that topic) the method is more likely to choose the areas we would like to swap out. Unfortunately, identifying the least recently used page takes a lot of work, and – as every reference to an area must be taken into account – at least some of the work must be done by the hardware. We'll describe two procedures which work.

  Perhaps the obvious way is to maintain a queue of areas which is reordered at every memory reference so that the most recently used area goes to the tail of the queue. The operation on the queue is quite simple and standard, operating on the page map or segment list, and involving no arithmetic. The list must be linked in both directions to eliminate time-consuming list traversals. The least recently used area is always instantly available from the head of the queue.

  An alternative is to number the references. The processor maintains an instruction counter (that is, a *real* counter which counts instructions executed, not the instruction address register), and stores the current value in a field reserved for the purpose in the page descriptor or segment header at every memory reference. That's much easier than reordering lists; but now each time the replacement strategy is used it's necessary to search through all the entries to find that with the smallest number.

  Clearly, both of these algorithms will work; equally clearly, both involve additional work for which special hardware is essential, and both use memory to maintain the ordering information. We shall see in the next chapter that the additional memory required can be of concern; it amounts to enough memory to hold a memory address to hold the backward link (for the first method) or an instruction counter value (for the second method) for each area. The instruction counter must be big enough not to overflow between system restarts – which means, for a system with one instruction each microsecond and providing for a ten-year run, around 48 bits.

- **Not Recently Used :** The complexity and expense of the least-recently-used strategy makes it unpopular, even though the idea behind it is perhaps the closest we can get to the ideal strategy. It's possible to make one step from the cyclic replacement strategy towards the least-recently-used strategy comparatively cheaply, and it seems to make a significant, and welcome, difference. This also requires additional space, but only to the extent of one bit per area, which we shall call the *used bit*. This bit is initially set (to ensure that areas are not swapped out immediately they are allocated); all the used bits are cleared from time to time, and set whenever the area is addressed. The strategy is to swap out an area with the used bit clear. While that is obviously far from perfect, particularly just after the bits have been reset, it introduces a bias towards swapping out areas which have not been used for a while which seems to be effective in practice.

- **Oldest Inhabitant** (or **FIFO** – First In, First Out) **:** This strategy is fairly easy to implement. As each area is brought into memory it is attached to the tail of a queue, and candidates for swapping are taken from the head of the queue. There is some overhead for queue insertion and removal (both on swapping out again and on releasing the area), but this is small in

comparison with the cost of the associated action. Unfortunately, it isn't a very good method, for any area which is very useful will be likely to remain in memory for a long time and eventually, useful or not, be swapped out.

By juggling used bits, counters, shift registers, and lists, you can evolve a large number of variations on these strategies. Our examples have covered guesswork (Cyclic), cheap first approximation (FIFO), good but expensive (LRU), and compromise (NRU) techniques; the others fall at various points around the compromise level.

- *whether changed* – if an area hasn't been changed since it was last read from the disc, we don't need to write it back again. To use this as part of the replacement strategy is not necessarily a good idea; it might save a little time in the addressing fault, but it says nothing about how useful the area is. It is more useful to avoid wasting time wherever possible once a selection has been made by some other means. To implement this, the page or segment descriptor must contain a marker (often called a "dirty bit") which is reset when the area is newly fetched from memory and set by hardware whenever anything is written into the area. It is also necessary to keep both memory address and disc address in the table; you can't economise by keeping only one.

- *whether locked in memory* – many systems provide means to mark selected areas as "locked in memory", which mean that they are exempt from swapping. This is sensible if the areas are known to be very frequently used, or their contents must be accessible very quickly. Addressing tables may be (though they are not necessarily) locked in memory; file buffers are also sometimes locked, on the grounds that it's silly to read something from the disc and then swap it out again. This requires a "locked bit" in the area descriptor.

None of them is perfect; some – particularly the uselessness guesses – require additional administrative overhead in normal running. They also need more information in the process's addressing table; the table entry might include some selection of a dirty bit, a used bit, a locked bit, a link for the list of pages or a 48-bit number, and others we haven't discussed. While it's impossible to guarantee the best choice of an area to swap out, it does seem that even fairly simple techniques can improve performance significantly above random choice – and even random choice will keep the system going !

IS IT REALLY AS EASY AS THAT ?

Sometimes. If you run just one process at a time and have a single processor, a system based on those ideas will work well. On the other hand, descriptions of the memory management algorithms are all very well, but it's far from easy to understand how they work without direct experience. It is particularly difficult to imagine how they work in a system where several processes might be operating simultaneously, and competing for the available memory. We therefore present a very contrived example (because a more realistic one would have to be run through many more cycles before anything interesting happened) which shows something of the possible interactions.

We assume that two processes, P and Q, run concurrently, executing programmes PrP and PrQ. The programme code is locked in memory, and can be ignored for purposes of memory management. To speed up events and generally simplify the behaviour, we assume that four pages of memory are available for the data, and data areas are not shared. Both programmes use the same code :

```
programme Pr*;

A, B, C : onepageofdata; { Data areas, each occupying exactly one
                           memory page. }

repeat
     use A;     { Short operation on variables only in A. }
     wait;      { Await interrupt. }
     use B;     { Short operation on variables only in B. }
```

```
        wait;     { Await interrupt. }
        use C;    { Short operation on variables only in C. }
        wait;     { Await interrupt. }
    forever;

    end of programme.
```

At each `wait` instruction, a programme waits for an interrupt (so we don't have to worry about details of instruction execution times); each programme has its own series of interrupts which never arrive at the same time (so we never have to worry about conflicts). The period between successive interrupts is much greater than the time taken for a page fault (so there's plenty of time for the virtual memory system to respond before another interrupt arrives), which is in turn much greater than the time taken to execute the `use` operations (so the speed of the programme is dominated by the page faults if any happen). We shall finally assume that none of the pages required is initially in the memory, so all sequences begin with four page faults which fill the memory. After that, it makes sense to compare the three examples of behaviour.

We shall investigate the frequency of page faults with both LRU and cyclic page replacement strategies under different conditions, defined by the relative rates of arrival of interrupts for the two processes.

Case 1 : The frequencies of P and Q interrupts are equal.

The page requests are in this order :

1	2	3	4	5	6	7	8	9	10	11	12
PA		PB		PC		PA		PB		PC	
	QA		QB		QC		QA		QB		QC

The allocations : page faults are in bold characters.

Cyclic			
page 1	page 2	page 3	page 4
1:PA	**2:QA**	**3:PB**	**4:QB**
5:PC	**6:QC**	**7:PA**	**8:QA**
9:PB	**10:QB**	**11:PC**	**12:QC**

LRU			
page 1	page 2	page 3	page 4
1:PA	**2:QA**	**3:PB**	**4:QB**
5:PC	**6:QC**	**7:PA**	**8:QA**
9:PB	**10:QB**	**11:PC**	**12:QC**

The frequency of page faults is the same as the frequency of interrupts for both processes. It is always necessary to use five other pages between successive uses of the same page, so neither algorithm can preserve a page long enough for reuse.

Case 2 : P interrupts are twice as frequent as Q interrupts.

The page requests are in this order :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
PA	PB		PC	PA		PB	PC		PA	PB		PC	PA		PB	PC	
		QA			QB			QC			QA			QB			QC

The allocations : page faults are in bold characters.

Cyclic			
page 1	page 2	page 3	page 4

LRU			
page 1	page 2	page 3	page 4

1:PA	**2:PB**	**3:QA**	**4:PC**
5:PA	7:PB		8:PC
6:QB	**9:QC**	**10:PA**	**11:PB**
		14:PA	16:PB
12:QA	**13:PC**	**15:QB**	**18:QC**
	17:PC		

1:PA	**2:PB**	**3:QA**	**4:PC**
5:PA			8:PC
	6:QB	**7:PB**	
		11:PB	
9:QC	**10:PA**		**12:QA**
	14:PA		
13:PC		**15:QB**	**16:PB**
17:PC	**18:QC**		

For the CYCLIC algorithm : The frequency of Q page faults is the same as the Q interrupt frequency. Five different pages are required between successive calls, so there is no chance that a page can be reused. The frequency of P page faults is only one half of its interrupt frequency. There are alternately three and four other pages required between successive calls, so half the time the required page is still there.

For the LRU algorithm : The performance is again exactly the same as that of the cyclic algorithm, though the sequence is different. The rather poor performance of this algorithm is a consequence of the iterative structure of the process, for which the least recently used segment is quite likely to be the next required – so several times a page is replaced immediately before it is required again.

Case 3 : P interrupts are three times as frequent as Q interrupts.

The page requests are in this order :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PA	PB	PC		PA	PB	PC		PA	PB	PC		PA	PB	PC		PA	PB	PC	.
			QA				QB				QC				QA				QB.

The allocations : page faults are in bold characters.

Cyclic			
page 1	page 2	page 3	page 4
1:PA	**2:PB**	**3:PC**	**4:QA**
5:PA	6:PB	7:PC	
8:QB	**9:PA**	**10:PB**	**11:PC**
	13:PA	14:PB	15:PC
12:QC			
	16:QA	**17:PA**	**18:PB**
19:PC	**20:QB**		

LRU			
page 1	page 2	page 3	page 4
1:PA	**2:PB**	**3:PC**	**4:QA**
5:PA	6:PB	7:PC	**8:QB**
9:PA	10:PB	11:PC	**12:QC**
13:PA	14:PB	15:PC	**16:QA**
17:PA	18:PB	19:PC	**20:QB**

For the CYCLIC algorithm : The frequency of Q page faults is the same as the Q interrupt frequency. Five different pages are required between successive calls, so there is no chance that a page can be reused. The frequency of P page faults is again one half of its interrupt frequency. There are alternately three and four other pages required between successive calls, so half the time the required page is still there. (The other half of the time, it has just been overwritten !)

For the LRU algorithm : We win at last – or, at least, P does. After the initial period, P needs no page faults at all, because the comparatively infrequent Q interrupts always come after all the P pages have been used, leaving the single resident Q page as the least recently used. Q therefore still requires one page fault for each interrupt – but, then, it always did.

What, if anything, do we conclude ? Not a great deal, because of the artificial constraints placed on the example – but these were intended as simplifying constraints, so we might expect that the real behaviour is more complex. What is very clear is that the simple patterns suggested by the names of the algorithms – cyclic replacement, least recently used – interact with execution patterns (in this case, the programme loop) and competition between processes in non-trivial ways, so that the actual performance of the algorithms is hard to predict.

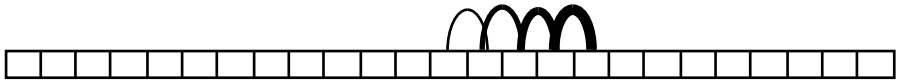
The consequences of these interactions go further than curious paging behaviour; they can extend to the design of the page management algorithms themselves. With a single process running in the system, once an addressing fault occurs not much can happen until the required memory has been brought from the disc. If other processes run concurrently, this is no longer true; circumstances can change while the rather slow machinery of the fault grinds along. For example, consider an attempt by a process to use a portion of its memory which happens to lie in an area marked by the replacement strategy for swapping out. The situation is clearly undesirable : the area which the replacement strategy guessed would be not wanted turns out to be very wanted ! What should be done ? The one course of action not open to the system is to permit the reference in the hope that everything will be all right; in a simple system, there is no way to tell how far the swapping operation has been completed, so the condition of the area of memory is unknown. On the other hand, if access to the area chosen for swapping out is blocked by setting its present bit to show the area absent as soon as the selection is made, a process which could continue is unnecessarily blocked, and it might send its request for the area to be swapped in from the disc before it has been swapped out. Another possibility is to abandon the original swap if the area is required in this way, and run the replacement algorithm again for the original process – but to do that, there must be some indication in the system that a reference to an area scheduled for swapping out has been made. The simple present bit is no longer adequate; we require at least the states *present*, *present but scheduled for swapping out*, and *absent*. (And if segments can be shared between processes, we also need the state *absent but already requested*.) (Observe that some of this information must be visible from the page map, as it affects operations outside the process.) The problems can be overcome, but the swapping software becomes more complicated.

WHAT HAPPENS – LOCALITY OF REFERENCE AND THE WORKING SET.

At every memory reference, the hardware checks the present bit; if the area isn't present, an addressing fault is signalled, typically as some sort of software interrupt. We saw earlier (in the chapter *MEMORY MANAGEMENT : THE PROCESSES' VIEW*) that we could describe the memory demands of processes in terms of locality of reference and the working set; in the context of virtual memory, the working set is important, as it establishes the current memory requirement of the process.

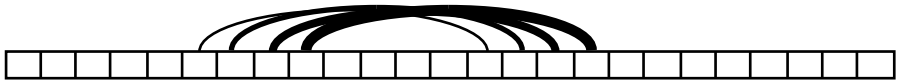
We can illustrate this principle by discussing the behaviour of some familiar sorting algorithms. We emphasise before starting that our comments have nothing whatever to do with the mathematical complexities of the algorithms, except insofar as they might affect the proportionality constants; we are concerned only with memory management questions.

Consider first a bubble sort. We begin with a set of numbers in an ordered list (usually implemented as an array), and the sort proceeds by successive passes through the list, with each step comparing the values of a pair of adjacent items and interchanging them if they are out of order. The diagram (like the two others which follow below) illustrates the process by identifying the memory references made in four sequential operations. Passage of time is denoted by the thinning of the markers, so the *thick* markers denote the *most recent* actions in the three diagrams which follow..



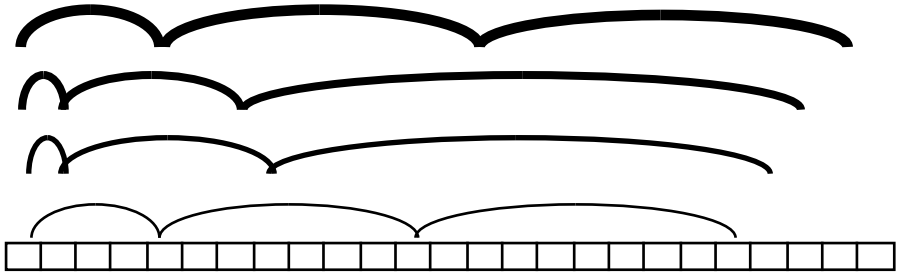
Everyone knows that a bubble sort is a very unsatisfactory way to sort items – everybody, that is, except the memory manager. For memory management, the bubble sort is close to ideal : the working set for the data is determined by the array references, and they are always to adjacent memory items. Except when the scan crosses a page boundary, therefore, the working set is likely to be close to one page in size.

Now consider the Shell sort. Again, we begin with an array of numbers, and the algorithm makes several passes comparing and perhaps exchanging values. The Shell sort differs from the bubble sort in that the items compared are not adjacent except in the later passes.



Now the working set in the early stages of the algorithm might be two pages in size.

Finally, consider a simple tree insertion sort. The items to be sorted are presented one by one, and inserted into the tree as new nodes with the sort order embodied in the tree structure. A new item is first compared with the root value, then directed to the left or right subtree according to the result of the comparison; and that procedure is repeated until a comparison leads to the construction of a new terminal node. The root value, as the first to be entered, occupies the first position in the storage array; the new nodes are always entered at the end; and the several probes in between will select intermediate points not randomly (as the later points, representing nodes lower in the tree, will be visited less often) but typically with a widespread coverage.



For this algorithm, the working set is likely to be large, as many pages are touched for each value inserted. This phenomenon is not restricted to sorting algorithms; an interesting account of its manifestation in various types of event queue, with experimental measurements, has been published[EXE19].

HOW TO MINIMISE SWAPPING.

Talking about reducing the number of addressing faults is easy enough; actually doing it is something else again. Here are some possibilities to explore; only the last even approaches a generally practicable solution which doesn't make unreasonable demands on the people who use the system.

- Take care with programme structure.
- Give the programmer control over segmentation.
- Make the compilers cleverer. (In a paged system : make sure that all the material in a page belongs to the same working set.)
- Buy more memory. (Or run fewer processes.)

WHAT'S HAPPENING TO VIRTUAL MEMORY.

People are less conscious of virtual memory now than they used to be. As the only reason for them to be conscious of it before was when it misbehaved (see the chapter on *THRASHING* later), that means that they are no longer noticing its misbehaviour. There

are a few reasons for this, and it's interesting to explore them briefly if only to speculate about the future of virtual memory.

The first change is that memory is cheaper, and therefore memories are bigger. That affects you directly if you're using a personal computer. It means that you can get your large programme into memory without using virtual memory (or, by the same argument, overlay methods), and you can get your large data structure into memory all at once.

If you are sharing a computer, you might still benefit from the cheaper memory. Other things being equal, you are likely to be able to occupy a greater area of primary memory than used to be common, so you can work with a bigger working set than in earlier years. So far as code goes, that might not help you very much; unless your programme is a curious special case, locality of reference still works just as it did before, and reasonably well structured programmes shouldn't need large code working sets. The greater freedom might help you with data, though you are only likely to benefit significantly if there are several areas of data to which you make quite frequent reference. Once your programme begins to use the virtual memory, you might not be much better off than in earlier systems.

A second significant change has been the increase in processor speed. If your programme is reliant on virtual memory, a faster processor might make very little difference. On the other hand, the faster processors lead people to expect more processing, which is likely to increase demand for memory.

So the picture remains unclear. Virtual memory systems are now available for microprocessors, and still in use on shared machines. The first paragraph of this chapter still seems to be true, and we don't think that virtual memory is on the point of disappearance. If anything, it might become more important, as it's one way to deal with the large memory spaces which are becoming available – perhaps the file systems will disappear instead.

COMPARE :

Lane and Mooney[INT3] : Chapter 11; Silberschatz and Galvin[INT4] : Chapter 9.

REFERENCE.

EXE19 : D. Naor, C.U. Martel, N.S. Matloff : "Performance of priority queue structures in a virtual memory environment", *Computer Journal* **34**, 428 (1991)

EXE23 : E.P. Markatos : "Visualizing working sets", *Op. Sys. Rev.* **31#4**, 3-11 (October, 1997).

QUESTIONS.

Where can you store information needed to implement (for example) least-recently-used virtual memory management strategies in paged memories ? Why can't you take a few bytes from the start of each page, as is common in segmented systems ?

What happens as a procedure is entered in a system using paged memory management ? Consider the significance of the segments.

Is there a relationship between the effectiveness of the simple not-recently-used replacement strategy and the idea of the working set ?

Under what circumstances would it make sense to permit addressing tables to be swapped out like any other memory area ? What happens when a page table is swapped out ?

How would you implement the first three suggestions we listed under the "HOW TO MINIMISE SWAPPING:" heading ?

The algorithm below transposes a matrix A. Suppose that each row of the matrix (row i includes the elements { A[i, j], 1 ≤ j ≤ N }) is a segment, what happens if the algorithm is run in a multiprogrammed segmented virtual memory system ?

```
var A : array [ 1 .. N, 1 .. N ] of integer;

i, j, t : integer;

for i := 1 to N
do   for j := i to N
     do   begin
          t := A[ i, j ];
          A[ i, j ] := A[ j, i ];
          A[ j, i ] := t;
          end;
```

(Note that quite different behaviours can be expected for different values of N.)

Will there be any changes in behaviour if the same algorithm is run in a system with paged virtual memory ? If so, what ?

What is the programme we've assumed in drawing the diagram for locality of reference ? Work out what would happen with other programme structures. What sorts of structure give bad localisation ? Are these likely to happen in practice ?
