

MEMORY MANAGEMENT : THE SYSTEM'S VIEW

In the previous chapter, we concentrated on the requirements of the programmes and explored some ways in which these could be met. This is very proper, and thoroughly consonant with our principle that the operating system's function is to offer computing services to entities requiring those services. Now, though, we move another level downwards, and ask what the system must do to support the services we have undertaken to provide; this is a good example of the client-server model in action, with the processes (or, if you like, the API procedures used by the processes) as clients and the system memory manager as server.

The server's job is to provide raw memory as requested. The request might be for a chunk of specified size in a segmented system, or for one page in a paged system, but in either case the system's memory manager must be able to respond appropriately. In order to do this, it must keep track of available space : there must be a *page map* or *available space list*. (Both do the same job, but are appropriate to the different management methods.) This is quite separate from the addressing tables : each process has an addressing table, which is essentially a device for linking its pieces together into a single addressing space; but the system's page map (or whatever) is used to identify the unused parts of memory.

PAGING.

A paging system uses a page map, with an entry for each page in the primary memory. This works because the number of pages in memory is fixed, so it's possible to allocate memory space exactly for the map itself. The page map can be quite small, as all we really *need* to know about each page in order to manage memory allocation is whether it is in use or available – so just one bit per page is enough. In practice, it might be helpful to remember more; for example, if the system knows which process owns each allocated page, then it can reclaim a page if a process owning the page ends without explicitly **releasing** it (perhaps because it has a corrupt page table). If this reclamation doesn't happen, the phenomenon called *memory leakage* can occur; we discuss this a little more later. Ownership information is also necessary if the system is to charge for memory use. Looking to the future, it's also useful to maintain some information used by the virtual memory system. Typical memory maps might contain a few bytes for each page.

While we want to carry all the information we need to make administration possible and effective, we do not want to waste space. Even a medium-sized computer system these days might have several thousand pages of memory, so a large record for each page can be quite expensive. We therefore would not usually include information which would only rarely be of use. For example, we've seen that, if we want to share pages between processes, it's helpful to have a usage counter; would it be sensible to keep that in the page table instead of the in-use-or-not indicator, so that every page could easily be shared if required ? The answer turns out to be more complicated than you might expect. First, as only one bit is required for the indicator, while several bits at least will be needed for the counter, the obvious answer is that unless a large proportion of the pages will be shared (which is rather unlikely) one should not carry the counter in the page map record. Second, though, it is quite likely that the single bit will be managed as a separate variable, and will probably occupy at least one byte anyway – so one might be able to carry a usage counter permitting at least 255-way sharing, which is enough for most purposes, without wasting any space at all. Third, though, as the fact that the page is shared is likely to be significant for any memory management operation, it will be necessary in any case to link the page to the system's list of shared memory areas, so there is plenty of room elsewhere for the usage counter, and it is more appropriately kept with the other information about sharing. We might therefore end up with an extended page map record anyway (to cater for the link to the shared table), but with the usage counter kept with the shared-memory information !

As well as fixing the number of pages, the fixed page size itself is easy to handle. There is only one sort of memory request – for one page – and the space vacated by a page which is no longer required is exactly the right size to hold any other page. These considerations might not strike you as astonishing; but compare the position for segmented systems, which we review below.

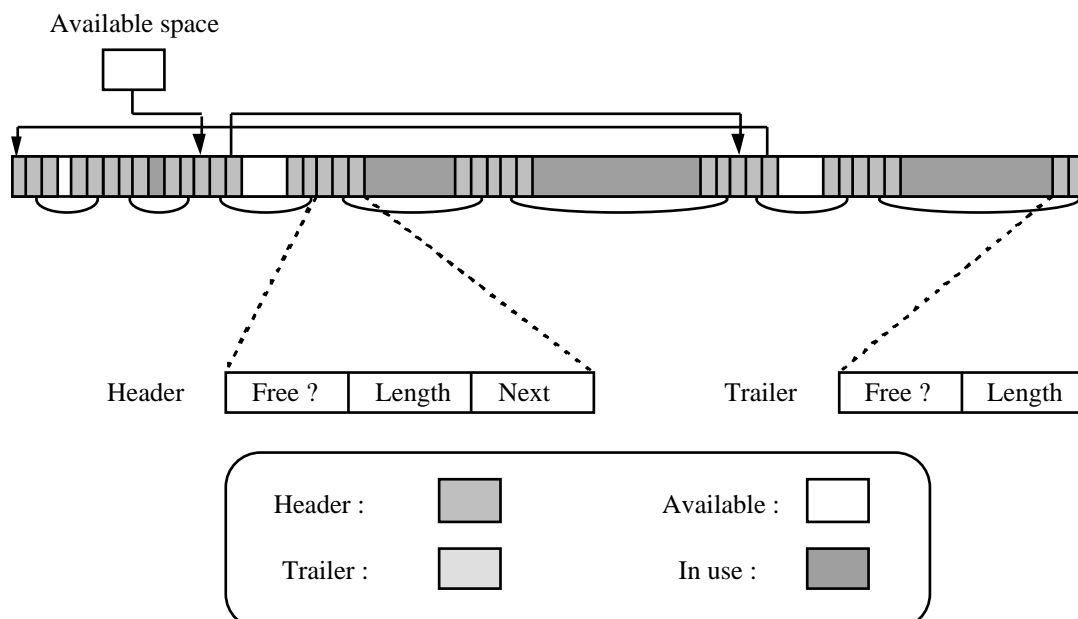
The other side of the coin is that the programmes' actual requirements for memory, not being of fixed size, do not fit the pages properly. Memory space is therefore wasted. Such wastage, which occurs within the allocated memory space, is called *internal fragmentation*.

A possible set of operations which might be implemented in a paged memory management system is listed below.

OPERATION	WHAT TO DO	
	In the (system's) page map	In the (process's) page table
get :	find a <i>free</i> page in the page map;	
	mark it <i>allocated</i>	
	return the page number	mark the virtual page <i>allocated</i> in the page table, insert the actual page number
release :	mark the page <i>free</i> in the page map	mark the virtual page <i>unallocated</i> in the page table

SEGMENTATION.

With a segmented memory system, a memory map of fixed size is less satisfactory, because the number of segments is not known beforehand; a fixed area would either be too big and waste space, or be too small and run out of slots. Instead a common device is to link together the available segments in a list, with useful information about the segment held in the segment itself. This useful information notably includes the segment size; this is a good place to keep it, because it is used by both the system memory manager and the memory access machinery, and if kept with the data it is easily accessible to both. Other useful information might include the segment's owner, which is useful just as it is for pages – for example, knowing the identity of the owner can help to avoid memory leakage. Here's a diagram of a typical (though atypically small !) segmented memory layout :



While there is no question of internal fragmentation with a segmented memory, as each allocation is of the requested size, the constant need to find available segments to

match unpredictable requests inevitably leaves unused space between used segments. This is *external fragmentation*.

The use of segments of different sizes in the memory makes it harder for the memory manager to find space as required. Instead of being able to use any space vacated by another programme, it must search through the available space list until it finds a vacancy of adequate size for the request, then allocate the appropriately sized chunk to the requesting programme, and return any significant surplus as a smaller available segment. It is this need for constant searching in the available space list which constitutes the biggest overhead for a segmented memory system. The Burroughs implementation of segmented memory was supported by specially designed hardware operators which would search linked lists in a single machine operation.

The task of searching for a vacant segment is peculiar to segmented systems. Algorithms for performing the task are called *allocation strategies*, and several varieties thereof are common. Here's a brief account of three examples

NAME	METHOD	REQUIRES	CONSEQUENCES
Best fit	Find the vacant segment which just satisfies the request.	Sorted free area list.	Tiny crumbs.
Worst fit	Always use the biggest vacant segment.	Sorted free area list.	No crumbs; no big vacancies either.
First fit	Take the first free segment you find that's big enough	Simple free area list.	Anything could happen.

The "crumbs" mentioned in those descriptions are tiny, and fairly useless, chunks of memory left over after allocating requested segments from vacant areas which are slightly too large. It is difficult to avoid such crumbs, and, once generated, they are very unlikely to be used. It is therefore all too possible to find that quite a lot of unused space is thinly spread throughout memory in the form of crumbs.

The obvious selection algorithm is perhaps the *best fit* method : choose the vacant segment which is closest in size to the request, thereby minimising the waste. At the same time, of course, it generates the smallest possible vacant fragment, which might turn out to be a useless crumb. We can avoid the crumbs, to some extent, by going to the opposite extreme, and using the *worst fit* strategy : always allocate from the biggest vacant segment. This avoids the crumbs, but destroys large vacant spaces, so it can be hard to satisfy a big request. To operate either of those strategies efficiently, it's necessary to maintain a list of available space ordered in fragment size, which can be time-consuming; we can avoid that by using the simple *first fit* strategy, where the system uses the first sufficiently large vacant segment it finds in searching sequentially through an unordered list of vacant segments.




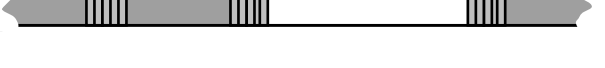



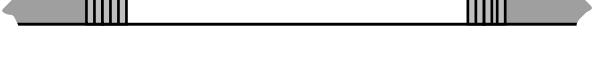
Which is the best ? Unless your system is struggling for memory most of the time, there isn't much difference. If your system *is* struggling for memory, the best answer is to buy some more; it's a better solution than relying on marginal improvements which might or might not accrue from a change in allocation strategy.

An interesting feature of the best fit approach is that in many systems exact fits are often found^{EXE17}. That's because the sizes of segments requested are a long way from randomly distributed. Most systems have some favourite memory sizes determined by characteristics of frequently used system components (they used to be 80 bytes, for punched cards, and 120 or 132 bytes, for line printers; now common sizes are determined by other devices and system

requirements) which are commonly requested and as commonly released.

If there is no sufficiently large vacancy, a memory manager for a segmented system might try moving segments about to coalesce vacancies. In practice, this is fairly slow, and the other processes can't run while you're doing it.

When processes release segments, they must be returned to the system in an orderly way so that the space is available for allocation. In general, the necessary manipulations follow from the specification of the memory structure as shown above. More specifically, there are four cases which must be managed, according to the configuration of the released segment, and they are described in the table below.

Before		The new vacant segment is isolated; it is linked to the available space list.
After		
Before		The new vacant segment precedes another; the existing vacancy is removed from the available space list, and reinserted with an appropriately changed length and position.
After		
Before		The new vacant segment follows another; the existing vacancy is removed from the available space list, and reinserted with an appropriately changed length.
After		
Before		The new vacant segment lies between two vacancies; the existing vacancies are removed from the available space list, and the single large new segment reinserted.
After		

We have assumed in this description that the available space list is kept in order of size, and that vacancies are coalesced as they are found. If the size order is not important, all cases but the first are simplified, as the originally vacant segment (or, in the last case, one of them) can be changed to denote the enlarged new segment rather than requiring removal and reinsertion. If vacancies are not immediately coalesced, the released segments can simply be attached to the list by whatever means are simplest; in this case, coalescence can be handled when segments are allocated, by checking for consecutive free segments if a free segment is too small to satisfy a request. Other variants are also possible.

This pattern of allocation, release, and coalescence leads to a curious consequence summarised in *Knuth's 50% rule*. This rule^{EXE18} asserts that, in a stably running segmented memory management system, the number of vacant segments is about half the number of occupied segments. Consider what happens when a segment is newly released; the release must follow the pattern shown in one of the four entries in the table above. In the first, the released segment is between two other occupied segments, so is in the

Centre of an occupied area; we call this (before release) a type C segment, and note that after release the number of vacant segments has increased by 1. In the second and third cases, each released segment is on the **B**oundary of an occupied block; it is a type B segment, and its release does not change the number of vacant segments. In the fourth case, the released segment is **A**lone, and must therefore be of type A; its release decreases the number of vacant segments by 1.

If we now denote the number of type A segments by a , and so on, we can work out the number of vacant and occupied segments, which we shall call v and s respectively :

$$s = a + b + c \quad (\text{because each allocated segment must be of one of the three types});$$

$$v = a + b/2 \quad (\text{because to each type A segment there correspond two vacant segment ends, and to each type B segment there corresponds one vacant segment end}).$$

(We have assumed that we can ignore end effects – which is equivalent to assuming that there are many segments in the system. (Or that the whole of memory is segmented and the system uses wrap-round addressing, which is unusual but not impossible.)) There is nothing in any allocation strategy which associates the position of an allocated segment with its contents or expected lifetime, so we can assume that the probability of a segment being released in any interval is the same for all segments, and that in consequence the probability that some type A segment will be released is proportional to a , and likewise for B and C. If we finally assume that the system has been running for some time and has reached a stable state, then, on average (we will expect some fluctuations, but we're concerned with the long-term behaviour) the rate of disappearance of segments must equal the rate of appearance of new segments – so

$$a = c.$$

From the previous equations (substituting for c in the first), we immediately deduce that

$$v = s/2.$$

That is Knuth's 50% rule.

There are several assumptions in that argument which can be questioned, not the least being that it is sensible to speak of a steady state – but the rule really does seem to work not too badly. It's interesting not only as a curiosity, but because it gives us some idea of the performance to be expected from a segmented system. Consider, for example, the table below. This is derived directly from the rule, and shows how the average size of the vacant segments depends on the number of segments (s) of average size () one hundredth of the memory size allocated.

s	Proportion of space occupied	Number of holes	Size of holes
10	0.1	5	18
20	0.2	10	8
30	0.3	15	4.7
40	0.4	20	3
50	0.5	25	2
60	0.6	30	1.3
70	0.7	35	<

Clearly, by the time the memory is 70% full, it will become fairly difficult to find a vacancy big enough to satisfy a request of average size. You can then either start moving segments about – which, as we have seen, is possible though rather slow – or you can buy some more memory. The table also shows that, up to about 70% full, you can reasonably expect to satisfy the average request for memory, and that really isn't too bad, for we would normally expect to run a system with a good margin for emergencies.

BUDDY SYSTEMS.

TERMINOLOGICAL NOTE : We take no responsibility for the nomenclature used in this description. We don't like it very much, but there it is. Perhaps you'll like it better than we do.

Buddy systems are among the earlier memory management techniques to be developed. They can cater for requests for chunks of memory of various sizes, but they do not rely on segments as we have described them; their chunk boundaries are fixed, but they do not rely on pages.

Instead, memory in a buddy system is organised as a binary tree of potentially assignable chunks. The whole memory is the initial chunk; at the next level, there are two half-memory chunks; then each of these can be divided into two quarter-memory chunks; and so on. The two chunks which result from each of these bisections are the buddies. Using this strategy, if the memory is 2^N units in size, there will be a chunk of size 2^M units for any $M \leq N$. The allocation strategy is to use the smallest chunk which will satisfy a request; but on release a chunk can only be coalesced with its buddy, even if there is another vacant block at its other side.

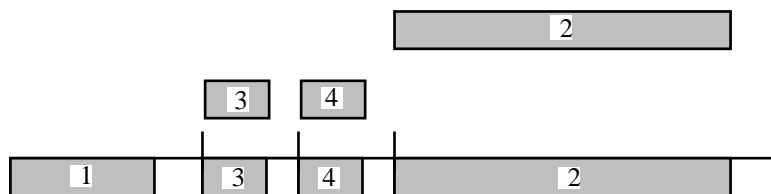
Here's a sequence showing how this works with a (carefully designed) arbitrary set of requests and releases. The process begins with an empty memory.



A request for memory is received. The request is for less than one quarter but more than one eighth of memory, so memory is divided first into halves, and then the first half is divided into quarters, and the first quarter allocated to satisfy the request.

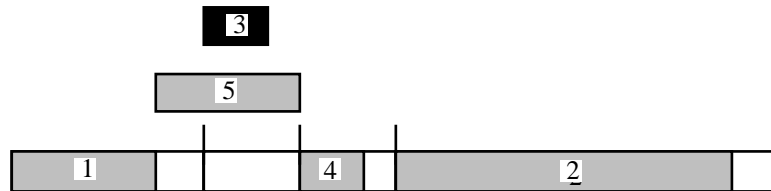


Three further requests can be satisfied by half of memory (there is already a vacant half, so that is used) and two eighths (so the remaining quarter is subdivided).

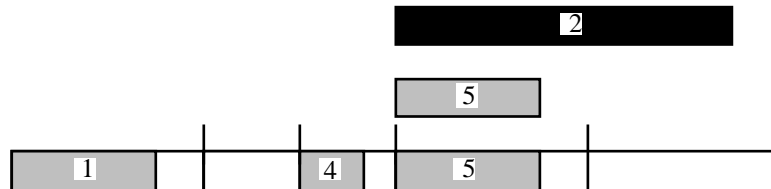


No more subdivision is now possible, even though a significant amount of free memory is available, because there is no vacant fragment corresponding to a proper 2^{-n} subdivision. (Some buddy systems will use the fragments if possible, regarding them as buddies of the parts of the areas already allocated. This is still in the spirit of the buddy system.)

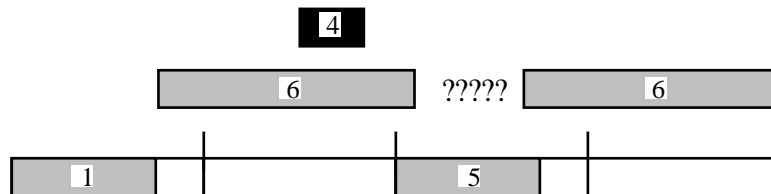
A small segment (black in the diagram below) is released, and a larger request is received. Although there is sufficient contiguous space to accommodate the larger segment, no allocation is made, because the two areas which make up the space are not buddies, so may not be coalesced.



The largest occupied segment is now released; the outstanding request can now be satisfied by bisecting the second half.



Another fragment is released, leaving well over half the memory free – but a quite modest request for well below half of memory cannot be satisfied without violating the buddy rules.



AVOIDING WASTE.

Waste of space (usually called "fragmentation") is unavoidable in any practical general-purpose memory management system – just as it is in any operation which requires that many things of odd sizes be packed into a given container. Mathematically, finding an optimum solution is known to be difficult, so memory managers, which have to solve the problem perhaps several times per second, don't try; they aim at a fairly good system.

PAGING : take care with the page size (when designing the hardware !) – there's a tradeoff between internal fragmentation and the space occupied by the page map. Once the page size has been determined there's very little that can be done to avoid waste. A clever compiler can try to pack code and data into pages – but if you want pure code segments, and shared code or data, that can impose severe constraints on the compiler's freedom. We discuss the optimum page size further in the chapter *PAGE AND SEGMENT SIZES*.

SEGMENTATION : choose a good allocation strategy (first fit, best fit, etc.). If segments are allocated and released without any forethought, it's easy to end up with quite a lot of useless crumbs. The phenomenon is sometimes called "checkerboarding", from the appearance of a diagram in which memory is represented as a sequence of strips with used segments shaded.

BUDDY SYSTEMS : these systems are designed to avoid crumbs, by always coalescing fragments to avoid little bits left over. Buddy systems don't abolish waste – if anything, they're worse than some others – but they do ensure that you don't end up with an impossibly "checkerboarded" memory.

One can argue that memory is so cheap nowadays that the waste doesn't matter. We are not convinced. We would urge that in engineering terms any waste is bad in principle, and to be avoided if possible. We can reasonably use arguments from cost to choose between methods that are available; presumably, therefore, even cheaper methods would be even better ! We also remark that the arguments from cost must be used properly, taking into account *all* the costs, not merely the trivially obvious. If your programme takes up 10% more space than it need, there is that much space which is not available for running other programmes, for expanding your own programme, and other purposes. In

a virtual memory system, every additional page used is an addition burden on the disc channels, and might significantly reduce the speed of execution of the programme. Economical use of memory might not be the vital necessity which it was when 64K memories were thought large, but it is still true that careless use of memory can hurt, and it is certainly appropriate that the memory management of the operating system should be carefully designed.

LEAKAGE.

A large operating system might carry out many thousands of memory management transactions every hour, and be expected to keep running for weeks or months on end. Obviously enough, every one of those transactions had better be right, or unfortunate consequences might ensue. One such consequence is *memory leakage*, by which we mean an apparent slow decrease in the amount of memory available to the operating system.

The cause is usually that some memory areas are being allocated, but never returned to the system pool. There are several ways in which this can come about, and the blame can lie with the programme which causes the leakage or with the operating system which fails to identify the leakage and take action to remedy it.

Programmes can cause leakage (or, more precisely, act in a way which is conducive to memory leakage) by allocating memory and then losing track of it. There are several ways to accomplish this feat : pointers to memory areas can be overwritten by pointer assignments, "dangling pointers" can be used out of scope, shared memory areas can be badly administered, programmes might contrive to corrupt their addressing tables, programmes might stop prematurely for one reason or another without tidying up their memory allocations.

But the misbehaviour of programmes should not be an acceptable excuse for failure of the operating system. It is the job of the operating system to keep the computer system going, whatever the processes which it oversees might try to do – so a good operating system will allow for the possibility that a process might behave stupidly and take precautions to ensure that the stupidity cannot affect other processes. The fundamental precaution is simple : never trust a programme. Use memory protection and supervisor calls to ensure that a process cannot reach any sensitive tables or other structures; maintain an independent record of all memory owned by the process; check that all the owned memory is released when the process ends, naturally or not.

The root cause of memory leakage is always a badly designed operating system. There's a lot of it about.

COMPARE :

Lane and Mooney^{INT3} : Chapter 10; Silberschatz and Galvin^{INT4} : Chapter 8.

REFERENCE.

EXE17 : R.R. Oldehoeft, S.J. Allan : "Adaptive exact-fit storage management",
Comm.ACM **28**, 506 (1985).

EXE18 : D.E. Knuth : *Fundamental Algorithms* – part 1 of *The Art of Computer programming* (Addison-Wesley, 1972), page 445.

QUESTIONS.

We suggested that keeping a segment's length with the segment data was better than keeping with the segment base address. Which is better if segments can be resized, or shared, or moved, or any combination of those ?

**What information must a system carry in order to avoid memory leakage ?
How would you implement leakage avoidance ?**
