PAGE AND SEGMENT SIZES

What determines the sizes of pages and segments in a memory management system ? (In practice, of course, the page size is determined by the processor hardware, but someone had to decide what size to build into the processor, so the question is still sensible.) The decision should be based on the performance of the processor under working conditions. It is far from easy to work out in the abstract just what these conditions will be, and how they will affect the decision. To study the performance seriously, simulators are often used, as no satisfactory sound mathematical treatment is known. Here we offer some very questionable statistics on segments, and some deeply unsound mathematics on pages which, like most such efforts, deals with a grossly oversimplified system to give totally unreliable answers. You should therefore place little weight on the numerical values which come out of the calculations, but the derivations do show that there are grounds for believing that an optimum size exists, and point to some of the factors which will affect its value.

HOW BIG ARE SEGMENTS ?

While segmented memory management is the rarer of the two varieties under consideration, we deal with it first because it is more directly tied to the structure of the work carried out by the computer; the segments are the chunks of code and data space which naturally behave as units. We shall also see that a knowledge of the segmentation properties of the workload has some bearing on the page size calculations.

We should be able to get the information we need to discuss the segments by inspecting the things which computers do – that is, by inspecting the programmes they execute. It turns out to be not quite as easy as that. We can look for measurements; not many seem to be available. Alternatively, we can try to argue from our knowledge of software structure and performance.

Measurements :

On a B5500 system : about 300 bytes. (Hearsay, ancient. The B5500 is an old Burroughs system with hardware memory segments. That isn't a deliberate choice – it's the only source we have. The information might be harder to get from systems which don't recognise segments.)

Arguments :

Consider code, file buffers, data structures, and other data.

Code : Structured programming folklore says that a programme module of any sort should be about a screenful of source programme – say up to 30 lines. Each line of source programme could correspond to about 10 to 20 bytes of code. (Measurements on a Cobol programme[EXE20] give 8 bytes per line, which is likely to be low given the typical wordiness of Cobol.) That gives 300 to 600 bytes.

(But recall that the small modules are only typical of fairly recent source programmes. In older programming styles, modules could be several thousand lines in length.)

File buffers : traditional file record sizes are 80, 120, and 132 bytes. There are plenty of bigger ones, but, in conventional data processing, it does seem to be fairly unusual to want to read more than a few hundred bytes of information about a single item. Databases might store many thousands of bytes associated with individual keys, but then different selections are commonly used at different times, and it makes more sense to split the information up into separate files, each containing only closely associated data.

(In other areas, this might not be so. In multimedia applications, for example, buffer space might be needed for temporary storage of streams of data, such as images or digitised sound. In such cases, though, as for memory used to support the screen display, the importance of having the space immediately available means that the buffers will be locked in memory.)

Data structures : Very large data structures exist, but they're almost always built up of repeating units – they're arrays of structures, or something like that, each of which is comparatively manageable. We hope (though we can't guarantee) that locality of reference will result in the lower level structures being used a few at a time.

(There will always be exceptions. We've mentioned the two-dimensional array problem and the awkward properties of tree searches as examples. Even these examples, though, don't really affect our arguments about segment sizes; if there are solutions, they lie in the area of memory replacement strategies rather than segment sizes.)

Other data : For the rest (vast linear arrays, large workspaces used by editors, etc.) we can make no prediction, except that they'll have to be split up somehow, so they'll just have to put up with whatever we decide.

The arguments, such as they are, are in gratifying accord with the observations, such as they are, and they lead to the conclusion that segment sizes are of the order of a few hundred bytes.

BUT SERIOUSLY NOW ....

Yes, we know that these numbers are not precisely up to date, and a lot has changed since the days of the B5500. Memories and programmes of many megabytes are now commonplace, and the idea of reading a complete, and quite large, file into memory to work on it is no longer absurd. But how much of the arguments have changed ? Not a lot, so far as the ordinary data processing is concerned. A new factor is the requirement for large blocks of memory used for special purposes – such as screen bit maps, or representations of smaller, but still considerable in terms of bytes, components of pictures. Commonly, though, such areas of memory are dedicated to their special jobs, and, once sliced out of the memory, are no longer involved in active memory management. Perhaps, then, our analysis will still hold, at least approximately (which was the most we ever claimed); in any case, we have not come across any more recent measurements, so it will have to do. Take this as a warning to look at the principles rather than the detailed results, though.

HOW BIG SHOULD PAGES BE ?

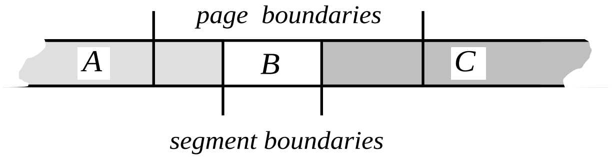
Notice the difference between this heading and the corresponding heading about segments. The size of a segment is in the nature of a natural phenomenon; it turns up as a consequence of the way we write the programme, and we don't normally control it directly. In contrast, the size of a page is quite independent of the programme, and it's reasonable to think of choosing a good page size as a part of our top-down design.

Coming back to the question, then, we wish to choose the page size that makes our system perform best. In what way does the page size affect the performance ? The only *direct* effect is in the amount of memory rendered unavailable for productive use (which is not quite the same as "wasted"); there are indirect effects, but we shall ignore them, thereby rendering the results null and void so far as quantitative significance goes.

The real wastage occurs at the end of each contiguously addressed chunk of memory : provided that the chunk is significantly bigger than the page size, we can think of each chunk as several full pages ending in a partly full page. The argument is based on the observation that with bigger pages you need smaller page tables, but the average fragmentation loss – presumably, about half a page for each contiguous chunk – is bigger.

We use the word "chunk" rather than "segment" because the area might be defined by criteria other than the structure of programme and data – for example, a simpleminded compiler might well compile a whole programme, code and data and all, into a single chunk, and the system then has no way to infer any segment structure that might be inside the chunk. In a segmented system, the chunks will be segments; some compilers can be prevailed upon to produce code in which there is some attempt to align certain structural features with page boundaries; if so, the areas so defined must be considered as chunks.

Combining the observations of the preceding two paragraphs, we find another, less obvious, sort of fragmentation which is often left out of account. The diagram shows three segments arranged in a flat memory – procedure code, data area, or whatever – and possible positions of page boundaries.



Now if it happens that only segment B is used by the programme, the page must be brought into memory, but the portions of the page occupied by A and C are quite unnecessary, and should be considered as waste. In a virtual memory system, the effect might still be felt even if A, B, and C are all used, if they are used at significantly different times. Indeed, the only occasion when the space is not wasted is when the three segments are used within the same working set.

It is clear that this phenomenon will only be of importance for segments which are significantly smaller than the page size. You might like to bear this observation in mind as the discussion proceeds.

There is yet another source of wastage which will be significant if virtual memory is not used. Without virtual memory, parts of chunks are of no use; no matter how clever our paging algorithm, we cannot fit five 13K chunks into 64K of memory. We shall not consider this problem further, as this form of wastage is important mainly when the chunks are sufficiently large that only a few can fit into the memory. As memories grow this condition is less often satisfied (and it would be satisfied even less if compilers for machines such as the Macintosh would make better use of the memory management facilities available instead of compiling enormous flat chunks which are treated as single segments by the memory manager). The problem remains significant in smaller microcomputer systems with limited provision for memory management; it is effectively solved in larger system by paged virtual memory..

SYMBOLS.

m	Memory size.
p	Page size.
t	Page table entry size.
c	Contiguously addressed chunk size (average).
n	Number of processes running.
l (length)	Total memory required by each process (average).
u	Unavailable memory.

HERE GOES.

Suppose to begin with that the system does use virtual memory. The total size of the page tables is the sum of the page table sizes for all the running processes; we therefore calculate :

Total page table size : nlt / p

From the argument above, the expected fragmentation loss is $p/2$ for each chunk in memory, of which there are m/c. Therefore,

Memory lost through fragmentation : $mp / 2c$

Adding the two areas of memory unavailable for normal use, we find the total unavailable memory :

$$u = (nlt / p) + (mp / 2c) \qquad (1)$$

To find the page size which gives minimum loss, we pretend that all the variables are real numbers (which we have been doing anyway, unless we suppose that division was implicitly redefined somewhere), and differentiate with respect to the page size :

$$\mathrm{d}u/\mathrm{d}p = (- nlt / p^2) + (m / 2c)$$

For minimum u, $\mathrm{d}u/\mathrm{d}p = 0$:

$$nlt / p^2 = m / 2c \qquad (2)$$

Notice that, at the minimum, the two terms on the right-hand side of equation (1) are equal. Equation (2) is equivalent to :

$$p = \surd(2\, cnlt / m) \qquad (3)$$

Combining (1) with (2) or (3), and rearranging to eliminate different variables :

$$u = mp / c = 2nlt / p = \surd(2\, nltm / c) \qquad (4)$$

SO WHAT ?

Once the machine design has been fixed, both m and p are constants defined by the hardware. From equation (4), the unusable space is therefore inversely proportional to c, the chunk size. Other things being equal, therefore, we would like to keep the chunks as big as we can. Unfortunately, even ignoring the problems with big chunks which we pointed out earlier, other things are not equal.

The processes must fit into the available memory, so $nl < m$, whence from (3)

$$p < \surd(2\, ct) . \qquad (5)$$

As it is advantageous to make the chunks as big as possible, we suppose that each process can be given a single chunk, whence $c < m / n$, and

$$p < \surd(2\, mt / n) .$$

For a fairly old system, we could imagine a half-megabyte machine running twenty processes with 8-byte page table entries : that gives $p < 600$ bytes. In practice, for obvious engineering reasons pages have sizes which are powers of 2, so for this machine one would expect 512 or 1024 bytes per page. It isn't very critical – if you work out some values, you find that the minimum is quite flat – but the order of magnitude is plausible.

What about a Macintosh ? Our experience suggests that with an 8-megabyte memory, the Macintosh system regularly complains about lack of memory if about five processes are resident in memory. The Macintosh memory management system uses segments, though these typically contain whole programmes rather than the natural segments which we have been discussing, and the system is by no means keen to move the segments about. In consequence, it is not unusual to have two "crumbs" of a megabyte or so, and still be unable to run a programme requiring 1.5 megabytes. We would quite like a paged memory which would resolve this silly situation. Assuming the same size of page table entry, and eight processes (realistic, but it simplifies the arithmetic too), we find a page size of about 4 kilobytes, which will give us a total unavailable memory of 32 kilobytes – certainly a lot better than the 2 megabytes of our example ! (Don't just believe all this – be critical. But, though the numbers might be suspect, we think that the ideas are sound.)

For a virtual memory system, we can't do the same trick, for several things have changed. First, notice that the page table for each process must provide for its total *virtual* size, as there must be an entry for every page whether in memory or swapped out. The total page table size remains unchanged at nlt / p, but the product nl can now exceed the size of the memory.

The total table size is a pessimistic estimate of the amount of memory which must be used, as it is possible to swap out the page tables; like any other use of memory, the absent pages will be reloaded when they are again required. This is another consequence of locality of reference; other things being equal, a working set is likely to be concentrated in a small area of a process's address space for significant periods of time, and during that time the page tables for other areas are not needed. We do not even attempt to take account of this possibility in our estimates !

It is still true that we want to use the largest possible chunks, but we can no longer give each process a single chunk, because we cannot now guarantee that the working set can be regarded as a single contiguous memory space – indeed, it is much more likely that it can't. What sort of chunks can we give ? They must correspond to parts of the programme which are contiguously addressable : some examples are the code of a single procedure, a single data structure, a file buffer. These are in fact the segments of a segmented memory manager.

First, we can try a chunk size typical of common segment sizes – say, 500 bytes : using (5), and the parameters of the original problem, we find that

$$p < \sqrt{(8000)} \approx 90 \text{ bytes.}$$

This is tiny; but it's obviously silly to have large pages with small chunks.

When we try to start up the virtual memory, we find that the situation changes. Suppose we try to run n programmes, each of size m. We must also increase t if we want the results to be at all comparable, as we now have to store a disc address and present bit in the page table. Suppose that $t = 12$. From equation (3),

$$p = \sqrt{(2\, cnt)} . \qquad (6)$$

With $n = 1$, we find a page size of 110, increased over the previous 90-byte value so that the smaller number of pages will reduce the page table size, compensating for the larger entries. But now, from equation (6), the page size increases as we increase the number of processes. This is again because, as the number of processes increases, they require more pages, so the page table – which includes pages not resident in memory – increases also; the increase in page size is again intended to reduce the size of the page tables. This eventually becomes absurd (which is why we have to swap page tables); if we welcome back our original 20 processes, the page size increases to 350 bytes, at which value about a quarter of the pages must be wasted in fragmentation. But notice that if we used the 600-byte pages appropriate for the system without virtual memory, none of the pages would be full !

CONCLUSIONS.

- To have pages bigger than segments is wasteful; so if the segments really are a few hundred bytes long, most paging systems must waste quite a lot of space.
- But if we try to use small pages to avoid fragmentation losses, we get enormous page tables, which force the page size up again. Luckily, we can avoid the worst consequences of this by paging the page tables.
- The real conclusion is that it is very hard to use memory efficiently, even when we really try.

Quantitatively, these calculations are certainly unreliable, but they do show something of the complicated interactions between different factors which make memory management a formidably difficult task to accomplish satisfactorily.

REFERENCE.

EXE20 : B. Meredith : *Online transaction processing systems*, Project report, Auckland University Computer Science Department, 1990

QUESTIONS.

Try to work out how the results derived in the chapter must be modified if other sorts of system behaviour are taken into account. For example, how does the page or segment size affect the cost of running virtual memory ?

Can you extend the "calculation" to allow for paged page tables ?

Can you extend the calculation to take account of the fragmentation resulting from page boundaries within contiguously addressed chunks ? You might try to work through the problem for the extreme case in which adjacent segments are never in the same working set.

We have assumed throughout that it is normal for closely related items (both code and data) to be placed close together in memory. This is a reasonably good assumption with the common model of computing. What happens if that assumption is false ? Are there forms of computing for which the assumption does not hold ? (Consider dataflow machines, neural networks, etc.)

In calculating the optimum page size, we assumed that it was sensible to allocate one memory chunk to a process. Many compilers which simulate stack models (Pascal, etc.) use this chunk by allocating static memory for code and data at the lower end, and implementing a stack growing downwards from the upper end. Assuming that all processes are organised in this way, what would be the effect of allocating two chunks to each ? (Note that the answer is *not* derived by simply setting $c < m / 2n$ in the equations.)
