THRASHING

Virtual memory is good when it works, but can be catastrophically bad when it doesn't. One important type of failure, called *thrashing*, might happen in a system perfectly designed and implemented to execute any straightforward memory management algorithm, as it can be caused purely by the behaviour of the running processes. It can happen when the sum of the sizes of the processes' working sets exceeds the size of available memory.

The principle can be illustrated by a very simple example. Consider a single process which makes reference cyclically, and rather quickly, to m different pages of memory. (Exactly analogous arguments work with segments, but pages, being more regular in size, are rather easier to think about.) Then its working set will be something like m pages. The computing required for each page takes time t, and the virtual memory system takes time T to fetch a page. Suppose that this process is executed in a computer with M pages of memory available. Then provided that $m \leq M$ all will be well, and, after an initial spurt of memory activity while the working set is established by setting up the m pages in memory, the process can run as quickly as the processor will let it. During the initial period, it will require time $T + t$ to complete the work on each page, but once all pages are present in memory only time t will be needed, so the complete cycle through the working set takes time mt. But what happens if $m > M$?

Suppose that $m = M + 1$. The execution begins slowly as with the smaller working set, with each page fetched from disc on request, taking time $T + t$ for each page. But now when M pages have been fetched, the last page is still not in memory. To fetch that page, it is necessary to swap out one of the resident pages – and the same necessity will arise at least once every cycle through the working set. The cycle now takes time $mt + rT$, where r is the number of page replacements which must be handled by the virtual memory, and depends on the page replacement strategy. The common least-recently-used strategy gives the worst possible result : $m(t + T)$. The best possible result is $mt + T$, which would be achieved by using the normally absurd "most-recently-used" strategy.

Obviously, the seriousness of the phenomenon depends on the ratio of t to T. If $t \gg T$, which would be so if a *lot* of computation happened within each page, then the problem would be negligible – but in that case we would regard the working set as a single page. By including the whole cycle in the working set, we have in effect declared that $t \ll T$, in which case even the best possible result is serious.

There is a very easy way to achieve such dreadful performance; it is the two-dimensional array example which we've already mentioned. You might have to adapt the details slightly to make it work with your language and compiler, but in general it goes something like this :

```
var matrix : array[ 1 : M, 1 : p ] of char; { p is the page size.
}

for index := 1, p
     for page := 1, M
          matrix[ page, index ] := matrix[ page, index ] + 1;
     end;
end;
```

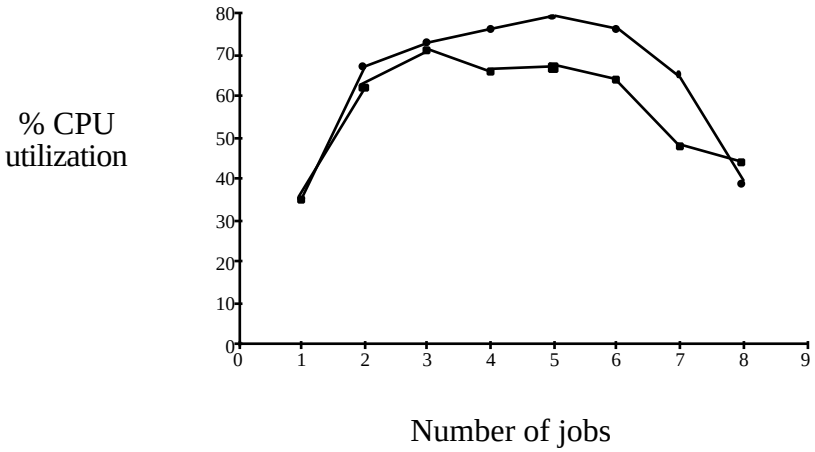
If it runs too quickly, try interchanging the two `for` lines.

Unfortunately, the example isn't a joke; many mathematical operations on large arrays scan the arrays in both directions at some time or another, and if you encode them in the obvious way it can take a very long time to get your answer. People who compose mathematical software worry about such behaviour, and develop techniques better suited to the behaviour of real memories. And it doesn't necessarily help just to buy another few megabytes of memory; memory might be cheap, but finding ways to soak it up is even cheaper.

That example shows how a single process running on a single processor can commit suicide, or something close to it. (Commit lethargy ?) That's fair enough : if you write silly programmes, you deserve to suffer. The really unfortunate fact is that, in a multiprogrammed system, other innocent people can suffer too.

Suppose we add to the system another process which is contained completely within one page. We shall call this the *nice* process. Just what happens now depends on details of timeslice length (see the next section) as well as the properties of the processes and memory which we've already used, so the behaviour becomes more complicated, but it's easy to see that thrashing is to some extent infectious. At the simplest level, in any system which uses a cyclic-replacement paging strategy, the nice process will be forced out of memory from time to time through no fault of its own. Even with a least-recently-used strategy, the nice process will be forced out any time that the nasty process happens to refer to each of its resident pages within a time slice before requiring an absent page. A quite nice process which uses just two or three pages in its working set is more vulnerable; and the situation becomes worse as more and more processes execute concurrently. Perhaps the worst case is that in which no process is identifiably at fault; all processes are reasonably nice, but there are simply too many of them.

The result of this sort of interference in the general case is shown in the diagram below. The experimental points (redrawn on this diagram) were determined[EXE21] from a simulator using two different models of the processing behaviour; clearly, they agree reasonably well, suggesting that the phenomenon is to some degree independent of details of the model. Notice that the maximum achievable processor performance is in the range 70% to 80%.



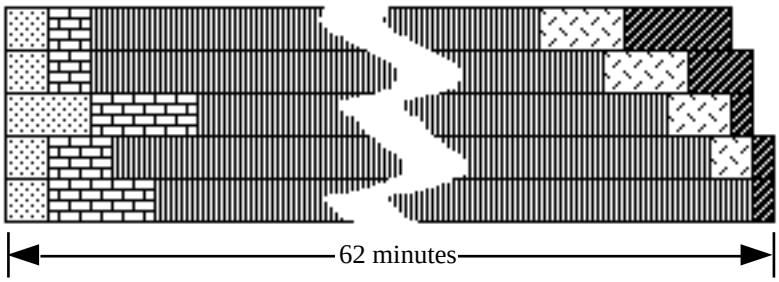
The infectious nature of swapping can be limited by imposing an upper limit of memory use on each process, and requiring that any additional demand be met by choosing the area to be replaced from the process's own memory. This works in principle, but is not very easy to implement sensibly in practice. It is necessary either to include ownership information in the memory map or to make provision for the replacement strategy to use the process's addressing table rather than the memory map. This becomes quite complex if there is to be any hardware component in maintaining lists used for the replacement algorithm; providing operations on a single predefined memory map is much easier than providing corresponding operations on addressing tables which might be of different sizes and might be anywhere in memory. A further complication is in finding a good way to fix the memory limit; it is clearly silly to fix a value, as that would force a process to swap against itself even in an otherwise empty memory, but to estimate reasonable values as the system operates is difficult.

AN EXAMPLE.

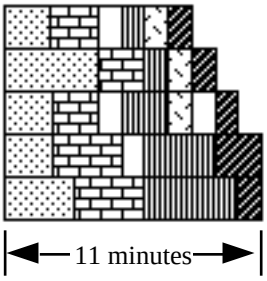
The University's first interactive computer service was based on a Prime P400 computer, with not enough memory or disc for the job it was expected to do. (We couldn't afford any more.) Thrashing was therefore common whenever a few people tried to run a big programme – such as a compiler – simultaneously.

The diagrams show how the performance of the system could be improved by constraining the compiler so that only one process could use it at any moment. The five jobs run in each case were identical : a typical file copy – edit – compile – load – run sequence. The timing information is all real; it was taken from a log file associated with each job. For reasons connected with the operation of the system, now lost in history, the times are accurate only to about the nearest minute at best.

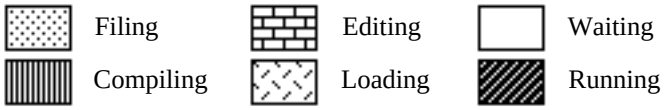
When the jobs were simply permitted to run together with no constraints, each job took about an hour, with the compiling step occupying most of the time :



When the system was changed so that only a single job could use the compiler at any moment, though, the pattern observed was quite different; the diagram below is typical. It is interesting that forcing jobs to do nothing for a while should get them all finished sooner !



The key for both figures is :



COMPARE :

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

REFERENCES.

SUP21 : S.E. Madnick, J.J. Donovan : *Operating Systems* (McGraw-Hill, 1974), page 505.