A time-wise hierarchy imposed upon the use of a two-level store.

Disclaimer. This note describes a way of looking at a class of virtual store implementations; the way seems illuminating and, as a result, virtual store implementations to which it is applicable may be expected to display some sensible characteristics. Not knowing all the --extensive!-- literature on the subject, I do not claim any novelty; Denning's survey article [1] of 1970 does not seem to mention it, not could I find it in the book by Coffman and Denning [2] of 1973 (nor in five other articles on demand paging and multiprogramming that at scanned).

Some simplifying assumptions about the hardware.

In order not to complicate the discussion unduly at the start I shall make a few simplifying assumptions. At the end of our discussion we may reconsider some of them; some are easily weakened, of others, however, we may come to the conclusion that, if our hardware does not allow such idealizations, the scheduling problem will be "complified" seriously, perhaps even beyond our comprehension and control. In the latter case we don't need to feel having failed "to cope with the problem", on the contrary: the identification of a seriously "complifying" factor seems in the light of the present state of affairs a valuable discovery.

As primary store we assume a store as randomly accessible as, say, a core store. As secondary store we assume a device with the characteristics of, say, a drum or a head-per-track disc, such that

1) place of information in secondary store need not influence decisions to change the contents of primary store

2) the processor speed is sufficiently slow and/or the cycle time of the primary store is sufficiently small and/or the transfer rate between primary and secondary store is sufficiently low that slowing down of the processor as a result of cycle stealing by the channel (to all intents and purposes) can be ignored.

3) a dedicated channel for the transport between the storage levels.

Furthermore I assume

4) demand-paging with fixed-size pages

5) a single channel for transport between primary and secondary store

6) a single processor

7) such a modest amount of processor-status information (registers included!) that the time needed for switching the processor from one process to another can be ignored in view of the upper bound on the frequency this switching has to take place

8) no page sharing between user programs (for instance possible on account of a common procedure library).

Note 1. The above assumptions are --or at least: were-- not unrealistic. All but the last one were satisfied in the case of the THE Multiprogramming System for the EL-X8. Had we known at that time Simon's article [3], the following considerations could well have been reflected in the design of that system in which, for instance, thrashing control has now been left to the operator.

Note 2. Usually the channel between the two levels of store is a semi-duplex channel, allowing a transport command either to read from or to write into secondary store. This results in the temptation

a) to keep track of "unmodified pages" in core store that need not be dumped because the original information is still on the drum, and, even worse

b) to allow the information which pages are still unmodified, to influence the replacement algorithm.

In the following I assume that this temptation does not exist --in a swapping drum with a full duplex channel it would be absent-- or --if such an ideal situation is denied to us-- it can be resisted. Our goal is "separation of concerns" and a coupling between the decisions what shall when be where on the one hand and the fact whether a page has been written into on the other seems contrary to that goal. (A milder form of yielding to the temptation is a separation between --constant-- program pages which are never dumped and data pages which are always dumped, even if in fact unchanged while residing in core. Still I would prefer hardware that made the distinction pointless, at least in the case of fixed size pages.)

<u>The role of the replacement algorithm.</u>

We assume a constant number Q of page frames in main store dedicated

to a particular program. Upon a so-called "page fault" --i.e. the desire to reference a page that is currently not in main store-- the missing page must be brought in from seconday store; parallel to that a page that occupied one of the Q frames has to be dumped. It is the task of the "replacement algorithm" to choose that victim, its goal is to keep the interesting pages in main store. Obviously, for each reasonable strategy, unreferenced pages have a tendency to disappear sooner or later from main store.

Unable to look forward, each replacement algorithm looks backwards, i.e. its reasonableness is justified by the assumption of some sort of continuity and the expectations regarding the near future are based upon the near past. (Very much like the whether prediction for tomorrow "Same whether as today.") Replacement algorithms differ in the amount of data they collect about past history and the way in which they extrapolate.

A well-known replacement algorithm records about the past history only the identity of the Q pages currently occupying the frames. When a victim has to be chosen, it makes --for lack of further information-- a (quasi) random choice. As a replacement algorithm it is not too bad.

Another one is called FIFO: the page that is in main core for the longest period of time is chosen as the victim. As a rule that strategy generates considerably more page faults than the random choice described above.

The best result have been achieved with LRU (Least Recently Used): here the pages in main store are permanently ordered (logically, not physically!) in the order of last reference to them and the one that has not been referenced for the longest period of time is chosen as the victim.

The LRU-algorithm, besides making the impression of being sensible, has a deep advantage: if the same program is re-executed with a larger value of Q, the number of page faults generated by this program shall not be larger. This property of monotonicity makes it very attractive. A disadvantage, however, is that the collection of the information --i.e. the ordering according to last reference-- is rather expensive.

A good compromise seems to have been discovered in the so-called "Second Chance Algorithm". Here only a boolean is associated with each page frame, a boolean which is set to true at each reference to the corresponding page. In the case of a page fault these booleans are scanned in a cyclic order: if the boolean is true, it is set to false and the scanning proceeds to the next one, if the boolean is false, the victim is found. At the next page fault scanning will start at the next boolean. Its ease of implementation and the fact that it is hardly noticeably worse than LRU have given the Second Chance Algorithm a well-deserved popularity. (I have described the Second Chance Algorithm in more detail because I do not know of a publication of it; I would love to give credit where credit is due, but this is one of those delightful small inventions that, by word of mouth, spread like wildfire.)

It is worth noticing that neither LRU, nor the Second Chance Algorithm, although reasonable, are always optimal. On the contrary: in the case of cyclic references to Q+1 pages, LRU is as bad as possible, as bad as FIFO. Instead of refining the replacement algorithm it seems in such a case wiser to detect the poor performance and to increase Q. Mind you, I am not advocating a brute force method --I am too much a puritan for that!-- but if we have the choice between spending our investment in a neutral facility tilke more core or in the implementation of a very refined replacement strategy, we should realize that there will always be reference patterns for which our refined strategy will fail miserably compared to other strategies. Some more primary store, however, will never do harm and will be useful under a greater variety of loads. Much of the extensive research spent on a great variety of replacement algorithms strikes me as rather ill-directed.

I honestly assume that something simple like Random or the Second Chance Algorithm will do, in spite of the fact that they fo not enjoy the property of monotonicity such as LRU.

<u>A program-wise balance between traffic time and computation time.</u>

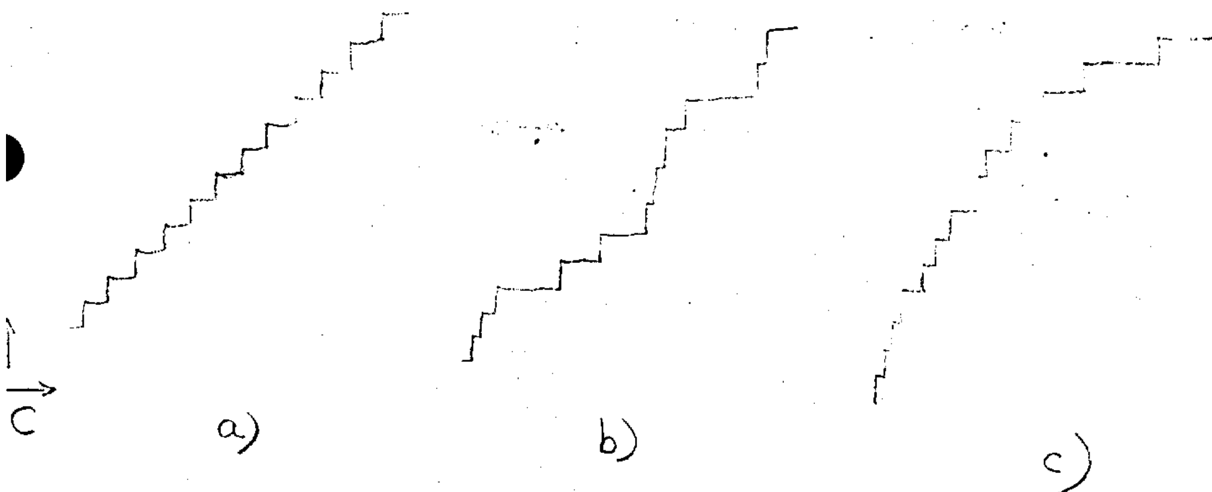
Let for a given program T be the total time the transport facilities between the storage levels have been monopolized on behalf of its page faults. Let for that same program C be the total time the processor has been monopolized on behalf of the progress of the corresponding computation. While

C is given by the computation to be performed, T depends on the chosen replacement algorithm and the number Q of dedicated page frames, the so-called "window size". (For the time being we assume that the window size is kept constant over the whole computation.)

When we now chose for each program a window size Q such that each program's T/C-ratio is about equal to 1, then each program absorbs the same fraction of the two resources "traffic" and "computation" and we can schedule them together as a single shared resource, assigning "priorities" as we please. Under what further assumptions would such system work reasonably well?

To start with, no program can ever utilize more than 50 percent of the combined resource and for full utilization we need therefore at least two programs. With two programs, full machine utilization is only possible provided the two programs generate their page faults with great regularity, for instance: during the first transport one idles and the other proceeds, which generates its page fault as soon as the idler can proceed. Such regularity is more than we can hope for, we must expect that the computaion times between successive page faults will differ. It is also clear what has to be done about it: increase the degree of multiprogramming. I have a gut feeling that with four or five programs simultaneously in the machine (and not too extreme scheduling) most of these irregularities will be absorbed. (The gut feeling is based on an old experiment; for a conservative two-particle system Boltzmann's distribution --which tells that for any particle at any moment of time the probability of a kinetic energy between U and U+dU is proportional to $\exp(-U/kT)dU$-- cannot hold; I simulated a three particle system and the exponential distributions came out beautifully! Admittedly the analogy is not too strong, yet it gives me the feeling that four or five is a reasonable guess; at a later stage we could try some baby-queueing-theory.)

For a given program we can we can plot the TC consumption: while C increases we move a pencil point to the right, while T increases we move a pencil point upwards. On the average our pencil point must rise under an angle of 45 degrees. Type a) is a program with small irregularities, type b) is a program with larger irregularities, type c), however, is a challenge to our assumption that a window size constant all through the computation is indeed a good thing: during the first half, which displays a T/C-ratio



greater than 1, we should like to decrease T, what may be achieved by increasing the window size; in the second half a smaller window size might bring the T/C-ratio closer to 1.

The next refinement, of course, will be to observe the T/C-ratios for the individual programs, decrease the window size when it is too small and increase it when it is too large. Compared with the target page fault frequency for a given program, the frequency of its window size adjustments must be an order of magnitude smaller, because --unless, perhaps, the window size is ridiculous-- the whole notion of a window size being adequate or not in view of the resulting T/C-ratio is only meaningful over a longer stretch of T and C consumption. Picture c) indicates that, if T and C are the accumulated consumptions since the last decision to set or keep the window size at its current size, the absolute value $|T-C|$ --i.e. the deviation from the diagonal--exceeding a threshold could be a good indication to reconsider the window size. (Note: if your target T/C-ratio is not 1, but 0.9, than one should take the absolute value $|T-0.9C|$.)

We may now hope to arrive at a very smoothly working system provided that for each program there is indeed a window size such that its T/C-ratio is close to 1. In the case of a program with highly interleaved references to 4 pages, howeverm a window size = 4 will give a T/C-ratio = 0 and a window size = 3 will give an arbitrarily high T/C-ratio. With window size = 4, we have T = 0 and we shall reconsider after C= threshold; if we then decide upon a window size = 3, then in the extreme case C will drop to zero and the window size is reconsidered when F = threshold, i.e. two such periods together have again a T/C-ration = 1 ! (This is just a joke,

not an intended feature. When I saw this, I started laughing. It says something about the stability of the window size adjustment mechanisms at this end of the scale and that is very encourageing. I think that I would keep the window size = 4 for longer periods of time: the window size oscillation with extreme T/C-ratios is not difficult to detect.)

In the case of a program with high vagrancy -a random referencing through 1.000.000 pages and hardly any computing at all-- we shall always have a very high T/C-ratio and if the system reacts upon this regrettable state of affairs by increasing and increasing its window size, then the system comes in a silly state that should be avoided. (At this side of the spectrum the window size adjustment mechanism suggested has not that stability.) It would be nice if for each program the system kept a list of the identities of the last so-many dumped pages: from this list we can derive a very good indication whether the most recent computation would have given a smaller T/C-ration with a larger window size. (If I am not mistaken, the guess can be absolutely accurate if the LRU strategy has been used for replacement; Second Chance Algorithm or Random seem to allow a good estimate.) We must be able to detect that increasing the window size is no good, yeah, even that, although the T/C-ratio is too high, a decrease of the window size might not make it worse; in the latter case the window size must tentatively be decreased until oscillation as mentioned above.

Change of window size will change the number of free pages -- when the number becomes too large or too small, we may or must decide to change the degree of multiprogramming.

* * *

The above has been written for many reasons. To a large extent it was like paying an old debt, repairing an old omission: we should have thought such type of thoughts a decade ago when we were designing the THE Multiprogramming System in which the replacement algorithm was LRU but then applied to the whole of main store. As a result a program with high vagrancy proceeded very slowly and could reside very long in the system.

I remember that we have considered for a moment to keep track of which page frames were used by which program but that we have given up

tp pursue this idea rather quickly, because we had difficulties with the shared library pages. What the difficulties were I don't remember, for now it seems obvious what to do. If program A requests access to a shared page that happens to be in the window of program B, the number of free pages increases by 1 because the two windos are made to overlap on that page. Both programs keep independently of eachother track of their own usage for the sake their own replacement algorithm. When the first of the two pushes the shared page out of its window the number of free pages decreases by one. When the last owner of the page pushes it out of its window, then the page disappears from main store. The possible gain of page sharing should not be too great, for otherwise the pressure to schedule such as to maximize such coincidences becomes too strong. Such pressure should be resisted, yielding to it would be terrible! (Immediately after I had written down and underlined this sentence, I found in Denning [4] "Allocation policies should tend to run two processes together in time whenever they are sharing information..." etc.!)

A second reason for writing the above down was an effort towards unmuddling my own mind. Last week the Dutch Computer Society NRMG organized a two-day Symposium on Operating Systems. (I spoke, but not at all about strategies; my subject was the mathematical relation between synchronization and sequencing, between deadlock and termination.) It was a depressing symposium: while most speakers wanted to talk about how to design a good operating system, the majority of the audience wanted to hear how to live with a bad one. And that is a different subject.... I was appalled during the discussion by the questions that came from the floor; I got the impression of operating systems that cause the machine's behaviour to be extremely sensitive to all sorts of workload characteristics, and, even worse, in a hardly predictable way. And I have the feeling that a machine is only useful provided the relevant aspects of its behaviour are reasonably well predictable and, also, as unsensitive to workload characteristics as we can possibly achieve. Such "smoothness" in the systems reactions seems much more important than the utmost efficiency, that justifies all complications that have been introduced. Honesty should even force us to add, that after all, efficiency considerations justify only the efforts and not the result. For from various participants I heard that they would regard a central processor utilization of 75 percent as a great achievement of the operating system! In what crazy world are we living? So I started to think.

The sketched approach --aiming both at smoothness and avoidance of extreme situations-- can be viewed as an effort to design with Simon's "The Architecture of Complexity" in mind. For the proper functioning of a hierarchical organization a necessary condition is that the natural grain for understanding each level grows by an order of magnitude when we go one level up. Here we are trying to understand what happens in time and our grains are therefore grains of time. The minimal time grain is of microseconds: the occurrence is a reference to a word in main store. This is applicable for the description of the happening during C-growth, between page faults. The next time grain is of the order of magnitude of a few centiseconds: at that level the relevant occurrence is the transport generated by a page fault. The next time grain will be of the order of magnitude of a second, viz. when for a program its window size will be reconsidered. The decision to change te degree of multi-programming will be taken with still lower frequency. On account of past experience I judge this as inspiring confidence.

The whole model gives me a better grasp on what you should say to someone who complains that he cannot keep his processor busy. If he is content with his throughput, you should say "Why bother?" or "What about a slower processor?" (In the early days I have seen, at an English University, where an IBM 7090 could not cope with the load, that this was replaced by an IBM 7094, so that that machine could wait four times as fast on its magnetic tapes!) If he bothers, faster store and transport will always do the job --this is, mathematically, of course the same suggestion as slowing down the processor. Just increasing the store will not help if the real source of the difficulties are the high vagrancy programs, it will help if he can increase his degree of multiprogramming in order to absorb the apparently severe irregularities. (Somehow I cannot expect that these severe irregularities in page fault occurrence can be successfully absorbed by a modest increase of the window size.)

Finally I have understood why I have never been able to see much significance in Denning's notion of the working set, viz. the set of pages referenced in the last t seconds of computation time: in that notion the intensity of the interleaving of the references is disguised, if not hidden and it is this intensity of interleaving which tells us how much the T/C-ratio might rise.

A final word about our simplifying assumptions. One channel and one processor are not essential, nor the absence of page sharing. Assumption 7 is not used explicitly in this note, it is of course essential in the realm of processor scheduling where you want to give the processor to the most urgen job and do not like to make this choice a compromise between the above desire and the desire to minimize the switchings from one to another. The fixed size pages are attractive for the simplicity of the considerations, to figure out how complicated this story would become for variable size pages (in the sense of "quanta of presence in main store") I just don't know. I guess that a similar story can be told.

Assumptions 1 and 2 seem pretty essential and so does number 3.

Finally, let there be no misunderstanding about this note: it has no scientific pretensions, nor should it be regarded as a suggestion for new implementations. It has been written for my own clarification and all others that would like to read this note are welcome to do so.

[1] Denning, Peter J., "Virtual Memory" Computing Surveys Vol.2, No 3 (Sep. 1970) 153 - 189

[2] Coffman, Jr., Edward G. and Denning, Peter J., "Operating Systems Theory", Prentice-Hall, Inc., Englewood Cliffs, 1973

[3] Simon, Herbert A. "The Architecture of Complexity", reprinted in Simon, Herbert A."The Sciences of the Artificial" MIT-Press 1968

[4] Denning, Peter J., "The Working Set Model for Program Behavior" Comm.A.C.M. Vol.11, No 5 (May 1968), 323 - 333 .

Author's address:
Burroughs
Plataanstraat 5
NUENEN - 4565
The Netherlands

prof.dr.Edsger W.Dijkstra
Research Fellow