Copyright Notice

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

by Edsger W.Dijkstra.

Authors address:
BURROUGHS
Plataanstraat 5
NUENEN - 4565
The Netherlands

Abstract: Following general design principles a paging system has been developed in which has been aimed at high efficiency, and a strong separation between store management and processor scheduling, and a minimal influence of the program mix upon the system's performance. It is, furthermore, described how some dedicated hardware can be expected to contribute effectively to memory management and the prevention of thrashing. Finally, the properties of the system should be such that a mismatch between configuration and workload gives a clear indication as to what reconfigurations seem indicated.

Key Words and Phrases: demand paging, window size, thrashing control, smoothness, virtual store, two-level store, operating systems, design, reconfiguration, separation of concerns.

C.R.Categories: 4.32, 4.34, 6.21, 6.34, 6.39.

NUENEN, 6th December 1974

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow

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

This paper is really two articles, merged into one. On the one hand it deals with a general design principle, on the other hand it deals with the design of a virtual storage system, a design to which that principle has been applied. Although the first aspect is the more general one, the title refers only to the second aspect, firstly because its elaboration occupies most of the space, and, secondly, because the virtual storage system to be developed below seems to be new and not without attractive properties.

The design principle in its most general form is that, whenever we have to design a mechanism meeting certain requirements, it does not suffice to design something of which we hope that it meets the requirements, on the contrary: we must design it in such a way that we can establish that it meets the requirements. As far as program correctness is concerned, this design principle has led to a programming methodology that is becoming more and more widely accepted: instead of making the program first and trying to establish its correctness afterwards --which may be near to impossible-- correctness proof and program are now developed hand in hand. (As a matter of fact, the development of the correctness proof is often slightly leading: as soon as the next argument in the proof has been chosen, a program part is designed so as to meet the proof's requirements.) Besides the mathematical requirement of correctness, we have the engineering requirement of "reasonable preformance" as well: this time the principle tells us, that it does not suffice to design a mechanism of which we hope that it will perform "reasonably well", but that we should (at least try to) design it in such a way that we can predict a priori how well it will perform. If we ask very precise questions about the performance, these questions may become very hard to answer: to predict that the computation time for the Horner scheme grows linearly with the degree of the polynomials is not hard; the estimation of the computation time needed for iterative computation of eigenvalues and eigenvectors of a symmetric matrix, however, is harder and probably most easily expressed in terms of the separation of the eigenvalues, i.e. in terms of part of the answer; then this dependance is something that we should try to derive and prove! Often we have to be content with "worst case" bounds (which in contrast to averages have at least the advantage of not depending on the usually unknown input population). Sometimes we have even to be content with still vaguer definitions

of what "reasonable performance" means: yet this is no licence to design, for instance, a mechanism whose performance is occasionally surprisingly bad.

The actual performance of a machine with a virtual storage system is dependent of what is usually denoted as "the workload characteristics". In the name of the predictability of that performance we shall try to design the system such as to make that dependence as simple as possible: in particular we require that a mismatch between configuration and workload does not only make itself manifest in the form of poor performance, but will in addition give a clear indication what type of change --if any-- of the configuration would improve the performance.

In order not to complicate the discussion unduly at the start, we shall make a few simplifying assumptions about the hardware. At the end we can reconsider these assumptions; some may be weakened easily, 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 seriously "complifying" hardware characteristics seems in the light of the present state of the art a valuable discovery.

As primary store we assume a random access 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, i.e. that page-wise it can be regarded as a random access 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 any slowing down of the processor as a result of cycle stealing by the channel (to all intents and purposes) can be ignored;

3) transport between the two storage levels is taken care of by a single, dedicated channel.

Furthermore I assume

4) a single processor

5) demand paging with fixed-size pages

6) such a modest amount of processor-status information (registers included!) that the time needed to switch the processor from one process to another can (to all intents and purposes) be ignored in view of an upper bound on the frequency with which these switchings may have to take place

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

Remark 1. The above asumptions are --or at least: were-- not unrealistic. We shall later discuss some of the temptations that should be resisted when they are only partly fulfilled. (End of remark 1.)

Remark 2. Assumption 6 means that as far as scheduling processor time is concerned, we can regard the total processor time as the sum of the periods of time devoted to actual program process, and are at any time free to grant the processor to what is concerned as the most urgent task. If the price of switching the processor from one task to another has to be regarded as high, one is faced with the often conflicting aim to grant the processor to the task with the maximum expectation value for the period of time for which full-speed progress is possible. (End of remark 2.)

The role of the replacement algorithm in a multiprogramming environment.

The idea of demand paging is that processing proceeds at full speed as long as the information is present in primary store. Upon a so-called "page fault" --i.e. the detected desire to access a page that is currently not in main store-- the missing page must be brought in from secondary store. (The program causing the page fault has to wait until the channel has completed that transport; in a multiprogramming environment the processor is in the mean time available for other programs.) Besides bringing in the missing page, another page has to be dumped. It is the task of the so-called "replacement algorithm" to choose that victim; its goal is to keep the interesting pages in primary store. Obviously, with each reasonable replacement algorithm, permanently unreferenced pages have a tendency to disappear sooner or later from primary store.

The ideal replacement algorithm embodies clairvoyance: it kicks out the page that in view of future needs can be missed best. Clairvoyance, however, is hard to implement, and actual replacement algorithms are based upon, essentially, three different ideas. (We shall later see that for our purposes the first two have to be rejected.)

1) With a (quasi-)random number generator an "arbitrary" page residing in primary memory is chosen as the victim. It is reasonable in the sense that permanently unreferenced pages have indeed a tendency to disappear from primary store, it is simple and its performance is not half as bad as might be expected.

2) In an effort to speed up the disappearance of permanently unreferenced pages the machine keepts track of the order in which the pages currently residing in primary store came in, and the older ones are given a greater probability of being chosen as the victim. In the extreme case, always the oldest is chosen and the algorithm becomes a FIFO ("First-In-First-Out") rule.

3) Predicting tomorrow's weather according to the principle "the same as today", the machine keeps to a certain extent track of the order in which pages currently in primary store have been accessed, and pages which for a relatively long time have not been accessed are given a greater probability of being chosen as the victim. In the extreme case we get the so-called LRU-algorithm ("Least Recently Used").

Note 1. In the case of cyclic access to $n+1$ pages with room for only n, both FIFO and LRU give the worst possible choice. As purely periodic access patterns are not unrealistic, it has been suggested to incorprate always a randomizing element in the page replacement algorithm, so as to reduce the probability of such a "disastrous resonance" to nearly nil. (End of note 1.)

We shall resume the discussion of the replacement algorithm later, because in a multiprogramming environment a more crucial decision has to be taken first. When a new victim has to be chosen, there are two alternatives::

1) either we regard primary store as a homogeneous pool of page frames and the victim is chosen on account of the total history in core, independent of the identity of the program that caused the page fault;

2) or we regard the page fault as a private occurence of the program in

which it happened, only the history of the pages of this program is taken into account and one of its own pages will be selected as the victim.

In the design of the THE-multiprogramming system in the early sixties I have chosen the first alternative and I remember the (opportunistic) arguments in favour of that decision: firstly it removed the obligation to keep track of which page frames were occupied by which programs --an administration that would have been complicated by the presence of shared library pages--, secondly it would automatically see to it that a program idling for other reasons would not continue to occupy page frames, as its then permanently non-accessed pages would disappear via the normal mechanism (which was LRU, related to the total history). This paper is a peccavi in the sense that --as I hope to demonstrate convincingly in the sequel-- this decision has been more than a mistake: it was a sin against proper design. (One of its unattractive features was that a large high-vagrancy program always lost its pages, and, as a result, suffered from very slow progress.) In the mean time we know that "separation of concerns" should be one of our dearest goals, and in the case of choice 1 the page faults caused by a single program are dependent both on its fellow-programs and on the relative speeds with which they are allowed to proceed. In the case of choice 2, however, were each program has its own, fixed number of page frames at its disposal, the generation of page faults is each program's private business, only dependent on that number of page frames, its access pattern and its(!) replacement algorithm. The mistake we made ten years ago was to allow a hardly controllable fine-grained interference between fellow programs that had been independently conceived but found themselves by accident mixed, instead of maintaining for these mutually independent programs to a much courser grain of time the mutual independency between their computational histories.

In the following we make the weak assumption about the replacement algorithm(s) used, that the average frequency of a program's page fault generation is a non-increasing (and usually even: a decreasing) function of its so-called "window size", i.e. the number of page frames allocated to it.

About the ideal window size.

In this section we shall describe how we propose to exploit our first three assumptions. After having observed that it is the function of the re-

placement algorithm to try to reduce --with a given window size-- the number of page faults caused by that program and, therefore, the total amount of time the channel is busy for the benefit of that program, our next purpose is to keep the channel nicely busy.

For each program we can introduce the total time C the processor has performed "computation" for that program, and also the total time T the channel has been occupied with "transports" between storage levels as a result of page faults caused by that program, both times C and T being recorded for that program since the same moment. When deciding how to allocate page frames to programs, i.e. when deciding the window size for each program, we seem to be managing three resources, viz. processor, channel and primary store. In this management problem, general dimension considerations tell us that the dimensionless quantity C/T must be significant. The point is, that processor and channel are resources doing something at a certain speed, but we cannot change the "speed" with which something is kept in store (no more than we are able to wait twice as fast for something).

Under the (temporary) assumption that for each program such a window size exists, we define for each program the "ideal" window size as the one that would give rise to a ratio C/T = 1 , i.e. the window size that would cause on the average equal demands on processor time and channel time, the reason being that then processor and channel can be scheduled as a single resource. The result of demand paging is that a program has no use for the processor during the period of time that the channel is busy for it; as a result no program can occupy more than 50 percent of this combined resource, and if we want to keep the latter busy, we conclude that our degree of multiprogramming should at least be equal to two. This degree will usually not suffice (see below).

About the degree of multiprogramming.

In this section we assume that for each program the vagrancy characterist are such that for each program a constant --and known-- window size can be considered as ideal.

In order to keep the combined resource constantly busy, individual C/T-ratios close to 1 is in general not enough. Suppose that the one program

generates its page faults --when executed all by itself-- quite regularly, one at a time, while the other program would generate under the same circumstances with half the frequency bursts of two page faults at a time: the combination would not fit and both processor and channel could be busy for at most 80 percent of the time. With a third program (of either type) full occupation is possible and an arbitrary program can use the maximum 50 percent. The typical purpose of multiprogramming is clear as far as utilization of the active resources is concerned: to absorb the bursts in which programs may generate page faults. After some consideration --and in analogy to other statistical phenomena-- it becomes hard to believe that the desire to absorb the bursts would ever give rise to a degree of multiprogramming exceeding 4 or 5 .

About the adjustment of window sizes.

We have introduced the notion of the "ideal" window size as the one by which program progress implies on the average equal loads C and T for processor and channel respectively. As a result the question whether for a given program the actual window has the ideal size or not, is meaningless unless it is related to a sufficiently large section of computation history, in which the increase of C + T is an order of magnitude larger than the T-increase caused by a single page fault (say: 20 times). Up till now, we have done as if during each computation the access pattern was sufficiently constant so that from beginning to end a single window size could be regarded as "ideal" for it, and also that for each program this size was known. In usual practice neither of these two conditions is fulfilled and, therefore, the system is required to discover for each computation what the ideal window size is, and to adjust for each program the window size when needed. For each program reconsideration (and possibly adjustment) of the window size should only take place with a frequency which is an order of magnitude smaller than that of the target frequency of page fault generation: it is pointless to be willing to vary a program's window size so rapidly that the periods during which it is by definition constant are so short that the question as to whether it was "ideal" becomes meaningless!

Let us assume therefore that for each program the system reconsiders its window size each time when that program has increased its C + T by a certain amount (equal to, say, 20 times the T-increase corresponding to a

single page fault.) When since the previous reconsideration of the window size C has increased much more than T , a smaller window might be more adequate, when T has increased much more than C , a larger window might be more adequate. We could think of a simple negative feedback, based upon the quotient of the observed increases of C and T , say decreasing the window size by one page frame when that quotient exceeds 1.1 and increasing the window size by one page frame when that quotient is less than 0.9 . Such a simple negative feedback, however, will not do the job, because even if our replacement algorithm is such that we can prove that a larger window would never lead to more page faults, the program might be such that a larger window would not lead to fewer page faults either!

A computation with high-frequency access to two fixed (program) pages and random access to 10,000 other (data) pages will not perform any better with a window of 100 frames (our maximum say) than with a window of 3 . If it has a window of 3 and its C/T ratio is too small, there is no point in increasing the window size. The simple negative feedback would continue to increase it and (like a young cuckoo) this program would eventually push the other programs out of primary store. This cuckoo effect cannot be remedied without penalty by suppressing growth of the window --although desirable on account of C/T-- as soon as no improvement is observed, and the reason is the following. A program with high-frequency access between 12 pages may perform equally poor with windows up to 11 frames and beautifully with a window of 12 frames, and this is something we would like to be discovered when its current window happens to be 4 . In other words: it is not enough to know the C/T- ratio caused by the current window size, we should also know it for other ones!

Monotonic replacement algorithms.

There is an important class of replacement algorithms --LRU is one of them, RANDOM and FIFO are not-- which we might call "monotonic", and are characterized by the following property. Considering two synchronized executions of the same program but with different window sizes, we call the replacement algorithm "monotonic" if at all times all pages contained in the smaller window will be contained in the larger window as well, provided that this was true at the beginning. As a result, in the computation with the larger window no page fault occurs that does not occur in the other computation as well.

Therefore, if a program is executed with a monotonic replacement algorithm and an actual window size w, it cannot cost much to record how many page faults would have occurred if the window size had been $w + 1$, $w + 2$ .... up to the maximum: it would only be a minor overhead on the actual page faults and would, therefore, be negligible. This information can be used to prevent the growth of a cuckoo, it does not cater for the detection of an existing cuckoo, i.e. a program whose window size can be decreased without any ill effects.

To record the page faults that would have occurred with window sizes smaller than the actual ones, additional hardware seems indicated. The knowledge of the number of page faults that would have occurred with smaller sized windows (particularly for the size $w - 1$) is so attractive to have, that the additional hardware seems justified. (In the latter case it can probably also take care of the recording of the number of page faults corresponding to window sizes larger than w.) Plotting page-fault frequency against window size it is not uncommon that this curve has a very sharp bend: we may expect programs that for a given window size w will give a ratio $C/T > 1$, while with a size $w - 1$ the ratio C/T would drop down unacceptably close to zero. With the simple feedback mechanism the effort at window size adjustment would lead to thrashing half the time --a nasty property of that feedback mechanisms that has been used as an argument against virtual storage systems as such--. If additional hardware counts the virtual page faults that would have occurred with window sizes smaller than the actual one, the thrashing half the time is easily avoided.

In view of the above it is doubtful whether the introduction of a randomizing element in the page replacement algorithm in order to avoid "disastrous resonance" --see Note 1-- is still desirable: most desastrous resonances occur when the window size is a few frames too small. But now we can detect this and know how to remedy it, it seems better not to obscure the detection by the noise of a randomizer.

The time-wise hierarchy.

At our lowest level we have the individual access: the recording of its having taken place (for the sake of the replacement algorithm) and the test whether it causes a (virtual or actual) page fault are obvious

candidates for dedicated hardware.

At the next level we have the actual page faults, which occur several orders of magnitude less frequently. Taken in isolation they only influence the program in which they occur.

At the next level, but again an order of magnitude less frequent, the window size is reconsidered. In the decision to increase or decrease the window size a threshold should be introduced so as to increase the probability that the result of reconsidering the window size will be the decision to leave it as it stands. Furthermore, if available information suggests a drastic change in window size, follow this suggestion only partly --half-way, say-- : either the suggestion is "serious" and the total change will be effectuated within two or three adjustments anyhow, or the suggestion is not "serious", because the access pattern is so wild, that the notion of an "ideal" window size is (temporarily or permanently) not applicable to that program. In the latter case it is better to allow this program to contribute unequal loads to the processor and the channel --if it only occupies one tenth of that combined resource, it can only bring the two total loads mildly out of balance-- .

At the last level, but again at a lower frequency, change of window sizes may have to influence the degree of multiprogramming: growing window sizes may force load shedding, shrinking window sizes can allow an increase of the degree of multiprogramming.

As a result of past experience, the fact that these different levels (each with their own appropriate "grain of time") can be meaningfully distinguished in the above design, gives me a considerable confidence in its smoothness, in its relative unsensibility to workload characteristics.

Efficiency and flexibility.

The purpose of aiming at C/T-ratios close to 1 was to achieve for the active resource (i.e. processor and channel combined) a duty cycle close to a 100 percent, to a large extent independent of the program mix. This freedom can still be exploited in various ways. A program needing a large window on account of its vagrancy can be given the maximum 50 percent

of the active resource in order to reduce the time integral of its primary storage occupation. Alternatively we can grant different percentages of the active resource in view of (relatively long range) real time obligations: to allocate a certain percentage of the active resource to a program means to guarantee a certain average progress speed. (This seems to me more meaningful than "priorities" which, besides being a relative concept, can only be understood in terms of a specific queueing discipline that users should not need to be aware of at all!)

Remark 3. When a producer and a consumer are coupled by a bounded buffer, operating system designers prefer to have the buffer half-filled: in that state they have maximized the freedom to let one partner idle before it affects the other, thus contributing to the system's smoothness. Granting no program more than 50 percent of the active resource is another instance of consicously avoiding extreme of "skew" system states! (End of remark 3.)

Temptations to be resisted.

If we enjoy the luxury of a full duplex channel, the page being dumped and the page being brought in can be transported simultaneously (possibly at the price of one spare page frame). Usually, however, such a page swap between the two storage levels takes twice as much time as only bringing in a page. If the channel capacity is relatively low, it is therefore not unusual to keep track of the fact whether a page has been (or: could have been) written into since it was lastly brought in: if not, the identical information still resides in secondary store and the dumping transport can be omitted. This gain should be regarded as "statistical luck" which no strategy should try to increase and which should never be allowed to influence one's choice of the victim (quite apart from the fact that it is hard to reconcile with the monotinicity of the replacement algorithm, as the monotonic replacement algorithm is defined for all window sizes simultaneously, independent of the size of the actual window).

We have also assumed the absence of page sharing. But this was not essential: if program A wants to access a page from the common library which at that moment happens to reside in program B's window, a transport can be suppressed by allowing the windows to overlap on that page frame.

Both programs keep, independently of eachother, track of their own usage of that page for the sake of their own replacement algorithm and the page only disappears from main store when it is no longer in any window at all. Again, this gain should be regarded as "statistical luck" which should never be allowed to influence our strategies. Such pressure should be resisted, yielding to it would be terrible!

Analyzing the mismatch between configuration and workload.

If the channel achieves a duty cycle close to 100 percent , but the processor does not, a faster channel (or more channels) or a slower processor may be considered. If the processor achieves a duty cycle close to 100 percent , but the channel does not, a faster processor (or more processors) or a slower channel may be considered. (With two processors and one channel each program has the target C/T-ratio = 2 .)

Note 2. A change in the quotient of processing capacity and transport capacity will give rise to other window sizes. With the built-in detection of virtual page faults as well, a user can determine himself what effect on the window sizes the change in that capacity ratio would have for his workload, without changing the actual window sizes. He should do so before deciding to change the configuration. (End of note 2.)

If neither processor, nor channel achieves an acceptable duty cycle, we either have not enough work, or are unable to buffer the bursts. If we have enough independent programs, a larger primary store could be considered so as to increase the degree of multiprogramming. Otherwise we should consider the attraction of more work, or reprogramming --so as to change vagrancy characteristics--, or a completely different installation (e.g. with very different secondary store characteristics). Or we may decide to do nothing about it at all and live with it.

Acknowledgements. Collective acknowledgements are due to the members of the IFIP Working Group W.G.2.3 on "Programming Methodology" and to those of the Syracuse Chapter of the ACM. Personal acknowledgements are due to the latter's Chairman, Jack B.Cover, to Steve Schmidt from Burroughs Corporation, to John E.Savage from Brown University and Per Brinch Hansen from the California Institute of Technology.