Copyright Notice

Monotonic replacement algorithms and their implementation.

(The following is written with demand paging for fixed-size pages in mind; the size of the pages being fixed is probably not essential.)

The idea of a virtual storage implementation is that not all the stored information (both program and variables) needed for the progress of a computation need to be in primary store simultaneously, but that for large periods of time parts of it may reside in secondary store. For this purpose the information is partitioned over a number of chunks such that during progress the information of a chunk will be either totally present in, or totally absent from primary store. In this sense the chunks are our "units of presence". If all the chunks have the same size, they are called "pages"; primary store is then subdivided into so-called "page frames", i.e. units of store able to contain exactly one page.

The idea of demand paging is that the computation can proceed at full speed until access to an absent page is required. Such a requirement is called "a page fault": the computation causing it comes to a grinding halt until the page needed has been brought in. If only pages were brought in, the capacity of primary memory would be exceeded very quickly; therefore, upon a page fault a page swap takes place: one of the pages present in primary memory while the page fault occurs is sent back to secondary store, is "dumped". The page subjected to this fate is called "the victim" and it is the purpose of the so-called "replacement algorithm" to choose the victim.

Elsewhere --in EWD462 (and in its preliminary version EWD408)-- I have argued that in a multiprogramming environment the victim should be chosen from the present pages of the program causing the fault. The number of pages that a program has present in primary store, its so-called "window size", is, as a result, not changed by the occurrence of a page fault. It is the purpose of this note to describe how the information is to be collected on account of which a reconsideration of the window size can be justified.

We call a replacement algorithm "monotonic" iff (i.e. if and only if) it has the following property. If the program were executed twice (but in strict synchronism) with two different window sizes, the pages present in the smaller window will at any moment all be present in the larger window, if this were the case at program start. Monotonic replacement algorithms

have the pleasant property that the page faults occurring with the larger window size are a subset of those occurring with the smaller window size, and an increase of the window size can never lead to a higher page fault frequency. It is easily seen, however, that a larger window size needs not to lead to a lower page fault frequency either.

Note 1. Here "frequency" is not meant as "number of times per unit of real time", but as "number of times per unit of computational time", i.e. with repect to a clock that runs while the program is being executed at full speed and is stopped while the computation is not in progress.(End of note 1.)

Note 2. In the sequel we shall take the freedom to consider for fixed window size the page fault frequency as a function of --computation, see previous note-- time, although a frequency cannot be the function of a moment, as it is only defined as an average over a period. For the time being we can think of something like

8 /(now - the moment of the last page fault but 7) .

Physicists --vide Lorentz-- do things like this all the time; we shall return to this later. (End of note 2.)

Although we know that at any moment the page fault frequency is a non-increasing function of the window size, we have without further information no knowledge about the slope of that curve (nor need, for a given computation that slope be constant in time). As a result, with a certain target page fault frequency in mind, we cannot trust the effectiveness of the simple feedback mechanism that increases or decreases the window size if the page fault frequency observed with the current window size was too high or too low respectively. (This would be like trying to keep a car on the road for which the actual steering mechanism reacts with unknown and varying sensitivity to a rotation of the wheel!)

In particular:

1) If the current window size gives a page fault frequency which is higher than the target value, we would like to know the larger window size (if any!) for which the page fault frequency would be small enough. (We just cannot expect to find this larger value by trial and error: if within

the bounds of primary store no such window exists, all trials become errors, and quickly even expensive ones!)

2) If the current window size gives a page fault frequency which is higher than the target value --and therefore, decreasing the actual window size is not something one feels tempted to suggest-- we would like to know how much the window size can be decreased without increasing the page fault frequency.

3) If the current window size w gives a page fault frequency lower than the target value, we should like to know the page fault frequency for a window of size $w - 1$: if that is much higher than the target value, we must abstain from decreasing the window size.

Note 3. The curve plotting the page fault frequency as a (non-increasing) function of the window size has very often rather sharp knees. In such a situation the simple feedback system can easily lead to thrashing half the time.(End of note 3.)

The moral of the above is that in order to justify an adjustment of the window size, we would like to know the (current) page fault frequency for all possible window sizes, and not just for the actual window size w. In the sequel we shall show how this information can be obtained for monotonic replacement algorithms.

Monotonic replacement algorithms define (independent of actual window sizes!) after each access a unique order for the pages of the computation that have been accessed at least once during program start. (In the following that ordering only interests us for the first $maxw$ elements, if $maxw$ is the maximum window size.) At any moment the k-th page in that order is the unique (!) page that would be contained in the window of size k , but not in that of size $k - 1$.

Consider now the effect of an access to a page which, prior to the access to it, is at position K in that order; upon completion of that access it must be at position 1 . (If we had executed the program with a window size $= 1$, the page concerned would have been in that single page frame window.) If $K > 1$, then the page originally in position 1 has

to move to a position higher up in the order, k_1 say; then the page originally in position k_1 has to move to a higher position, k_2 say, etc. until a page is brought into position K. More precisely:

with $k_{i_0} = 1$, $k_{i_n} = K$ and for $0 \leq j < n$: $k_{i_j} < k_{i_{j+1}}$

a cyclic permutation of pages has to take place with the page originally at position K moving to a lower position (viz. 1), all other ones moving to a higher position. For position k with $k > K$, the ordering remains unaffected.

Note 4. If, for $0 \leq i < K$ we take $k_{i+1} = k_i + 1$, i.e. each page originally at a position $k < K$ moves one position higher up in the order, we have the LRU-algorithm (Least Recently Used). For each window size w we have that $K > w$ indicates a page fault, the page originally at position w is indeed both the least recently used one and also the one that will be pushed outside the window. (End of note 4.)

Note 5. All other reorderings than the cyclic permutations described above would lead to more than one page moving to a lower position in the order, i.e. for some window sizes an unasked for page would be brought inside the window, but that is not what we call "demand paging": the combination of demand paging and monotonicity makes the above cyclic permutations the only permissible ones. (End of note 5.)

* * *

The mechanism consists of a string of mosquitos numbered from 1 through wmax. Mosquito nr. i has a variable cp (current page) whose value equals --for the moment we assume that the mosquitos are fast enough-- the name of the page currently in the i-th position of the order. Furthermore each mosquito is activated by placing a page name on its "A input" and one on its "B input". The A input will equal the name of the page that arrives in its position, the B input is the name of the page being accessed. Upon access of a page, its name is placed on both A input and B input of mosquito nr. 1. The code for mosquito nr. i is: (for LRU)

```
if cp ≠ B input → A output:= cp;
                  B output:= B input;
                  cp:= A input
 [] cp = B input → cp:= A input
fi
```

where the output of mosquito nr. i is the input for mosquito nr. i + 1 .

Left alone, the mosquitos will update their cp-value in the order of increasing ordinal number. If the accessed page was originally in position K , the first K-1 mosquitos will select the first alternative, the K-th mosquito will select the second alternative and there the "ripple" ends. If $K > w$, a genuine page fault occurs.

If this string of mosquitos were used to detect the presence or absence of a page, the transmission speed of the ripple would have to be very high viz. wmax mosquitos per memory access at least. Under the assumption of independent presence/absence detection with respect to the current window, higher mosquitos may lag behind! It suffices if they can go through the above motions with a speed of once per memory access: they are like the elements of a fancy shift register.

For the i-th mosquito each selection of the first alternative corresponds to a page fault that would have occurred if i has been the actual window size. Each mosquito has to extract from this series a corresponding "page fault frequency". They can do so by taking the past into account by an exponentially decreasing weight, for instance by keeping each a variable amppf ("average moment previous page faults") and transmitting "now", and adjusting each time the first alternative is selected amppf for instance by

$$amppf := amppf + (now - amppf)/8 \quad ,$$

(where "now" refers to the moment that the ripple entered the string of mosquitos). If for a certain window size the page faults occur at regular time intervals "delta", then in the limit:

before each adjustment: $now - amppf = 8 * delta$ and

after each adjustment: $now - amppf = 7 * delta$.

If we don't like this discontinuity, we can store per mosquito in addition for instance amppf' , each time updated by

$$amppf' := amppf' + (now - amppf')/2 \quad .$$

With page faults occurring at regular time intervals "delta", we have then in the limit:

before each adjustment:	now - amppf' = 2 * delta
after each adjustment:	now - amppf' = 1 * delta .
As a result we have constantly	amppf' - amppf = 6 * delta ,

and with the above we have achieved a Lorentz-like smoothing (see Note 2.)

* * *

Two questions have been left unanswered, but it seems premature to try to settle them now.

The first question is what to do when a processor switches from one program to another. As an elephant contains the information of wmax mosquitos, wmax may be high and processor switching may occur at great frequency, switching one elephant with equal frequency from one program to another might lead to unacceptable switching delays. I can only think of the crude solution: have at least as many elephants as we have high-priority programs. With LSI-techniques --the more of the same hardware, the better-- this is perhaps no so unacceptable as it sounds in my puritan ears.

The second question is how the collected information for a program is to be delivered. This has to occur at a page fault --when the victim has to be chosen-- and upon reconsideration of the window size. Particularly in the first case the "lagging behind" of the mosquitos higher up in the order presents some difficulties: it makes instantaneous selection of the victim impossible.

19th December 1974
NUENEN - 4565
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow