Two problems in connection with the LRU algorithm

0. Introduction

These days I am spending some of my spare time on the development of a so-called disk-cache program for my personal computer (, which fortunately is not a PC). The purpose of this program is to speed up accesses to the filing system by keeping copies of the most recently accessed disk sectors in a buffer in primary store. The assumption is that these sectors are likely to be accessed again in the near future; by keeping them in primary store many accesses to the relatively slow disk can be avoided. The price to be paid is the need to allocate a fixed part of primary store for the buffer; if primary store is so large that this part would otherwise remain unused, this modus operandi is viable.

A disk cache resembles a virtual storage system very much, but there is an important difference. In a virtual storage system the smallest action is an access to a single word in primary store, which requires very little time — in my computer: 0.5 μs —. In a disk cache the smallest action involves copying a whole sector, which requires much more time — in my computer: 340 μs, for a 512 bytes sector —. As a result, we can allow the disk cache to spend much more time on bookkeeping operations than would be acceptable in a virtual storage system.

I have decided that the replacement strategy needed for victim selection will be Least Recently

Used. Its implementation requires some amount of bookkeeping per "smallest action", which is the reason why the use of LRU in virtual storage systems is only feasible with the aid of dedicated hardware. (See, for example, EWD 465 in [0].) In my case, however, the overhead due to the bookkeeping can be kept within reasonable limits.

In this note I wish to work out in detail two aspects of the administration needed for the implementation of victim selection à la LRU. The solutions presented here are elaborations of a suggestion by L.A.M. Schoenmakers, for which he is kindly acknowledged. The two aspects are closely connected but can be discussed in relative isolation.

1. <u>Victim selection</u>

The first problem is to maintain a sequence x of values and to implement the following operations on x:

(i) extend x at its head with a given value
(ii) remove an "identified" element from x
(iii) yield the value of x's last element

Until Schoenmakers told me his suggestion, the only implementation I knew of was the one in which x is represented as a doubly linked list; this is feasible in that the above 3 operations can then be performed in constant time, but I never have been very fond of doubly linked lists (and I probably never will be).

A simpler representation is the one in which the elements of x are simply enumerated, in order, in an array y (say). If y has a fixed length then this is only possible under additional assumption of the invariance of, for some given constant M,

$$\#x \leq M \quad .$$

(In my application this assumption is valid.) In order to amortise the cost of the sometimes necessary shifting operations in array y, we decide to let y have a length that is substantially larger than M.

With N for the length of y and with $\perp$ for a value never occurring in x, we need an additional variable f and we use the following invariants.

P0 : "x is the subsequence of y's non-$\perp$ elements"
P1 : $0 \leq f \leq N$
P2 : $(\forall i : 0 \leq i < f : y.i = \perp)$

The three operations on x can then be implemented as follows.

```
(i)    if  $f = 0 \rightarrow$ cleanup $\{f > 0\}$
       ▯  $f > 0 \rightarrow$ skip
       fi
       $\{f > 0\}$
      ; $f := f - 1$
      ; $y.f :=$ "the new value to be added to $x$"
```

The precise specification of cleanup is:

$\{ P0 \wedge \#x < M \}$
cleanup
$\{ P0 \wedge P1 \wedge P2 \wedge (\forall i : f \leq i < N : y{\cdot}i \neq \bot) \}$,

where it is left understood that cleanup should not change x . Developing a program for cleanup is a straightforward exercise ; this program takes N steps . Because $\#x < M \leq N$ and $\#x = (\#i : 0 \leq i < N : y{\cdot}i \neq \bot)$, we have that $N-M < f$ is a postcondition of cleanup as well. Because f is decreased by 1 during every extend-x operation, cleanup is invoked at most once every $N-M$ extend-x operations. So, on the average the invocations of cleanup require $N/(N-M)$ steps per extend-x operation . For $N = 2*M$ this expression yields 2 .

(ii) It is sufficient (for my purposes) to let the element to be removed be identified by its index in y ; thus, we simply have:

$\{ 0 \leq p < N \wedge y{\cdot}p \neq \bot \}$
$y{\cdot}p := \bot$

(iii) In order to obtain a solution with, on the average, constant cost, we introduce a variable v with additional invariants:

P3: $f \leq v \leq N$
P4: $(\forall i : v \leq i < N : y{\cdot}i = \bot)$

In order to maintain P4 cleanup must be accompanied by $v := N$. Under the precondition that x is nonempty its last element can be computed by:

```
{ $\#x > 0$ , hence (by P0..4) : $(\exists i : f \leq i < v : y.i \neq \bot)$ }

; do $y.(v-1) = \bot \rightarrow$    $v := v - 1$  od
{ $y.(v-1) \neq \bot \;\wedge\; (\forall i : v \leq i < N : y.i = \bot)$ }
```

To investigate the (in)efficiency of this program we $\overset{need}{\wedge}$ an auxiliary variable h with invariant

P5: $\quad (\# i : f \leq i < v : y.i = \bot) = h$

P5 is maintained by accompanying cleanup with $h := 0$ and by accompanying $y.p := \bot$ in (ii) by $h := h + 1$. (Notice that by now $y.p \neq \bot \Rightarrow f \leq p < v$.) In the above program each statement $v := v - 1$ must be replaced by $v, h := v - 1, h - 1$. This shows that the number of steps taken by the above program is at most h ; and, by P5, h is at most the number of times operation (ii) has been invoked since the most recent invocation of cleanup. Thus, the execution time of operation (iii) can be amortised by attributing it piece wise to the preceding removal operations. (see P.S. on p.10!)

This concludes the solution to our first problem. The amortisation discussion in part (iii) can be formalised by the introduction of a so-called credit function, but the situation here is so simple that doing so is hardly worth the trouble.

I am very pleased with this solution and I find it definitely more elegant than the version with doubly linked lists. Moreover, it is another illustration of what I tentatively have dubbed the 50% rule. Consider the extreme cases $N = M$ and $N = \infty$. In the case $N = M$ storage utilisation is maximal and performance is almost zero, because cleanup may be invoked in every execution of operation (i). In the case $N = \infty$ storage utilisation is zero and performance is maximal, because cleanup is never invoked (, if we choose y to be an array with domain $i: 0 \leq i$ and reverse the order of x's elements in y). So, $N = M$ represents the optimal situation from the viewpoint of storage utilisation, whereas $N = \infty$ represents the optimal situation from the viewpoint of speed. The in between situation $N = 2 * M$ yields a fair compromise between these two extremes: storage utilisation is 50% —w.r.t. the extreme $N = M$— and performance will be approximately 50% —w.r.t. the extreme $N = \infty$—. The 50% rule states that, if we are able to obtain such a compromise, we have every reason to be satisfied.

2. Counting page faults

As has been observed by E.W. Dijkstra (EWD462 and EWD465, in [0]), LRU is monotonic in the following sense. From the viewpoint of virtual storage management a computation is a sequence of page names —called a reference string—. Consider the effect of a single, fixed, reference string upon the contents of the buffer —i.e.: the set of pages present in primary store— , as a function of the buffer size. If, at any point in the reference string, the set of pages in the smaller buffer is a subset of the set of pages in the larger buffer, and this holds true for all reference strings, then the replacement algorithm is said to be monotonic.

We can define, at any point in the reference string and for any page, the size of the smallest buffer containing that page; with a monotonic replacement algorithm, this <u>single</u> number identifies <u>all</u> possible buffer sizes for which an access to that page causes a page fault.

In order to be able to evaluate its performance and to obtain information about what would be a reasonable buffer size, I wish my disk-cache program to record, for each possible buffer size (upto a certain maximum), the number of page faults corresponding to that buffer size. In view of the above this boils down to the following.

We consider an array $y(i: 0 \leq i < N)$ whose elements are integers. Elements of y will be modified arbitrarily; nevertheless, it must be possible to compute $(\Sigma i: 0 \leq i < q: y.i)$, for every $q: 0 \leq q \leq N$. Simply computing the value of this expression takes $O(q)$ time, which is deemed too expensive. The other extreme is to store the values of this expression for all q, but this gives rise to prohibitively many updating obligations whenever an element of y is modified. So, as in the previous section, we are looking for a compromise.

How about a logarithmic solution? We define:

$$s.p.\ell \;=\; (\Sigma i : p \leq i < p+\ell : y.i) \quad , \quad 0 \leq p \leq p+\ell \leq N$$

Then :

$$\begin{array}{lll} (s0) & s.p.0 & = \; 0 \\ (s1) & s.p.1 & = \; y.p \\ (s2) & s.p.(k+\ell) & = \; s.p.k \; + \; s.(p+k).\ell \end{array}$$

It will be nice if we $\overset{can}{\smile}$ construct a set V of pairs (p,ℓ) , $0 \leq p \leq p+\ell \leq N$, with the following properties :

- Every interval $[0,q)$ is the disjoint union of $\mathcal{O}(\log.N)$ many intervals $[p,p+\ell)$, with $(p,\ell) \in V$. If the values $s.p.\ell$ have been precomputed and stored , this allows computation of $s.0.\underline{q}$ in logarithmic time .

- Every point $q : 0 \leq q < N$ occurs in $\mathcal{O}(\log.N)$ many intervals $[p,p+\ell)$, with $(p,\ell) \in V$. This restricts the number of updating obligations of values $s.p.\ell$ to $\mathcal{O}(\log.N)$, for every change of a (single) element of y .

We have :

$$\begin{array}{ll} & s.0.(q+1) \\ = & \quad \{ \; (s2) \; \} \\ & s.0.q \; + \; s.q.1 \\ = & \quad \{ \; (s1) \; \} \\ & s.0.q \; + \; y.q \quad , \end{array}$$

which is simple enough. In view of our efficiency requirement , we also consider :

$$\begin{aligned} & s.0.(2*q) \\ = \quad & \{ \ (s2) \ \} \\ & s.0.q \ + \ s.q.q \quad , \end{aligned}$$

which is not simple enough : there are too many of such summands. Therefore, we observe that $s.0.q$ and $s.0.(2*q)$ can be considered as instances of the more general $s.0.(a*q)$; we introduce a function f which we require to satisfy :

(f0) $\quad f.a.q \ = \ s.0.(a*q)$

The above problem can now be circumvented because :

$$f.a.(2*q) \ = \ f.(a*2).q$$

Furthermore :

$$\begin{aligned} & f.a.(2*q+1) \\ = \quad & \{ \ (f0) \ \} \\ & s.0.(a*(2*q+1)) \\ = \quad & \{ \ (s2) \ \} \\ & s.0.(a*2*q) \ + \ s.(a*2*q).a \\ = \quad & \{ \ (f0) \ , \ \text{introduction of function } g \ \} \\ & f.(a*2).q \ + \ g.a.(2*q) \quad , \end{aligned}$$

where function g is defined by :

(g) $\quad g.a.q \ = \ s.(a*q).a \qquad$, for a,q: $a*q+a \leq N$

By storing the values of function g , we obtain that $s.0.q$ can be computed in ${}^2\log.q$ steps , because we have :

$$\begin{array}{lcl} s.0.q & = & f.1.q \\ f.a.0 & = & 0 \\ f.a.(2*q) & = & f.(a*2).q \\ f.a.(2*q+1) & = & f.(a*2).q \ + \ g.a.(2*q) \end{array}$$

Moreover, we have, using (g) and (s1) :

$$g.1.q \quad = \quad y.q$$

Hence, the values $g.1.q$ need not be stored. The only values of parameter a that occur are powers of 2; for fixed k, the range of q in $g.2^{k+1}.(2*q)$ is :

$$0 \leq q \ \wedge \ 2^{k+2} * q \ + \ 2^{k+1} \ \leq \ N \ ,$$

which is satisfied by at most $N/2^{k+2}$ values q. As a result, the total number of g-values to be stored is at most N/2. (This bound is sharp: the actual number is N <u>div</u> 2.)

We conclude this section with an analysis of the updating obligations induced by a modification of $y.p$, $0 \leq p < N$. A stored value $g.a.(2*q)$ must be updated — i.e.: increased by $\Delta y.p$ — if and only if p lies in the interval $[a*2*q \ , \ a*2*q+a)$; this amounts to :

$$2*q \ = \ p \ \underline{\text{div}} \ a$$

This means that for fixed a we obtain the following program fragment that must accompany
$y.p := y.p + \Delta$:

```
  h := p div a
; if  even.h  →  g.a.h := g.a.h + Δ
  []  odd.h   →  skip
  fi
```

This program fragment must be executed for every a from those powers of 2 that also satisfy $2 \leq a \wedge a < N$. Hence, the number of update obligations is at most ${}^2\log \cdot N$.

)

3. P.S. to section 1

When rereading section 1 it suddenly dawned upon me that invariant P4 can be exploited to change the program for cleanup in such a way that it takes v steps instead of N steps. As a result, each decrease of v by 1 (in the program on p.4) saves 1 step in the cleanup operation. This offers an additional compensation for the cost of operation (iii), particularly so because in the LRU application, operation (iii) is always immediately followed by operation (i).

4. Afterthought

Let anybody who still believes that formal techniques for program development are only applicable to toy problems remain silent forever.

reference

[0] E.W. Dijkstra: "Selected writings on computing: a personal perspective", Springer-Verlag, New York, 1982.

Eindhoven, 14 april 1992

Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven