International Journal of Engineering Research and General Science Volume 2, Issue 4, June-July, 2014
ISSN 2091-2730

A Study of Page Replacement Algorithms

Anvita Saxena [1]

[1]Research Scholar, M.Tech (CS), Mewar University, Rajasthan

E-mail- anvita21saxena@rediffmail.com

Abstract-- A virtual memory system requires efficient page replacement algorithms to make a decision which pages to evict from memory in case of a page fault. Many algorithms have been proposed for page replacement. Each algorithm is used to decide on which free page frame a page is placed and tries to minimize the page fault rate while incurring minimum overhead. As newer memory access patterns were explored, research mainly focused on formulating newer approaches to page replacement which could adapt to changing workloads. This paper attempts to summarize major page replacement algorithms. We look at the traditional algorithms such as Optimal replacement, LRU, FIFO and also study the recent approaches such as Aging, ARC, CAR.

Index Terms- Page Replacement, Optimal Replacement, LRU, FIFO, ARC, CAR, Aging.

INTRODUCTION

The full potential of multiprogramming systems can be realized by interleaving the execution of more programs. Hence we use a two-level memory hierarchy consisting of a faster but costlier main memory and a slower but cheaper second memory.
In virtual memory the combined size of program code, data and stack may exceed the amount of main memory available in the system. This is made possible by using secondary memory, in addition to main memory [1]. Pages are brought into main memory only when the executing process demands them, this is known as demand paging.
A page fault typically occurs when a process references to a page that is not marked present in main memory and needs to be brought from secondary memory. In such a case an existing page needs to be discarded. The selection of such a page is performed by page replacement algorithms which try to minimize the page fault rate at the least overhead.
This paper outlines the major advanced page replacement algorithms. We start with basic algorithms such as optimal page replacement, LRU, FIFO and move on to the more advanced dueling ARC, CAR, Aging algorithm.

PAGE REPLACEMENT ALGORITHMS

A. Optimal Algorithm

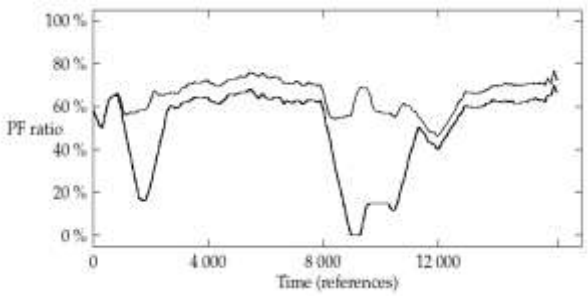
The Optimal page replacement algorithm is easy to describe. When memory is full you always evict a page that will be unreferenced for the longest time. This scheme, of course, is possible to implement only in the second identical run, by recording page usage on the first run. But generally the operating system does not know which pages will used, especially in applications receiving external input. The content and the exact time of the input may greatly change the order and timing in which the pages are accessed. But nevertheless it gives us a reference point for comparing practical page replacement algorithms. This algorithm is often called **OPT** or **MIN.**

B. Least Recently Used (LRU)

The LRU policy is based on the principle of locality which states that program and data references within a process tend to cluster. The Least Recently Used replacement policy selects that page for replacement which has not been referenced for the longest time. For a long time, LRU was considered to be the most optimum online policy. The problem with this approach is the difficulty in implementation. One approach would be to tag each page with the time of its last reference; this would have to be done at each memory reference, both instruction and data. LRU policy does nearly as well as an optimal policy, but it is difficult to implement and imposes significant overhead [3].

The result on scan data is as follows.

Algorithm	Ref count	Page count	Page faults	Hit count	Hit ratio
LRU	16175	7150	10471	5704	63.20%

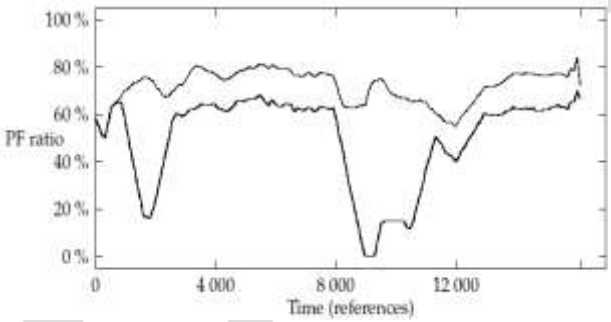


Scan datapage fault ratio using LRU

C. *First In First Out (FIFO)*

The simple First-In, First-Out (FIFO) algorithm is also applicable to page replacement. All pages in main memory are kept in a list where the newest page is in head and the oldest in tail. When a page needs to be evicted, the oldest page is selected and the page is inserted to the head of the list and the page at the tail is swapped out. Another implementation is using a ring (usually referred to as clock): Every time a page has to be replaced, the page the pointer points at is swapped out and at the same place the new page is swapped in. After this, the pointer moves to the next page. The FIFO algorithm's performance is rather bad [2].
The result on scan data is as follows :

Algorithm	Ref count	Page count	Page faults	Hit count	Hit ratio
FIFO	16175	7150	11539	4636	51.37%



Scan data page fault ratio using FIFO

D. *Adaptive Replacement Cache (ARC)*

The Adaptive Replacement Cache (ARC) is an adaptive page replacement algorithm developed at the IBM Almaden Research Center [4]. The algorithm keeps a track of both frequently used and recently used pages, along with some history data regarding eviction for both. ARC maintains two LRU lists: L1 and L2. The list L1 contains all the pages that have been accessed exactly once recently, while the list L2 contains the pages that have been accessed at least twice recently. Thus L1 can be thought of as capturing short-term utility (recency) and L2 can be thought of as capturing long term utility (frequency). Each of these lists is split into top cache entries and bottom ghost entries. That is, L1 is split into T1 and B1, and L2 is split into T2 and B2. The entries in T1 union T2 constitute the cache, while B1 and B2 are ghost lists. These ghost lists keep a track of recently evicted cache entries and help in adapting the behavior of the algorithm. In addition, the ghost lists contain only the meta-data and not the actual pages. The cache directory is thus organized into four LRU lists:
1. T1, for recent cache entries
2. T2, for frequent entries, referenced at least twice
3. B1, ghost entries recently evicted from the T1 cache, but are still tracked.
4. B2, similar ghost entries, but evicted from T2
If the cache size is c, then $|T1 + T2| = c$. suppose $|T1| = p$,

then |T2| = *c* - *p*. The ARC algorithm continually adapts the value of parameter *p* depending on whether the current workload favors recency or frequency. If recency is more prominent in the current workload, *p* increases; while if frequency is more prominent, *p* decreases (*c* - *p* increases).
Also, the size of the cache directory, |L1| + |L2| = *2c*.
For a fixed *p*, the algorithm for replacement would be as:
1. If |T1| > *p*, replace the LRU page in T1
2. If |T1| < *p*, replace the LRU page in T2
3. If |T1| = *p* and the missed page is in B1, replace the LRU page in T2
4. If |T1| = *p* and the missed page is in B2, replace the LRU page in T1
The adaptation of the value of *p* is based on the following idea: If there is a hit in B1 then the data stored from the point of view of recency has been useful and more space should be allotted to store the least recently used one time data. Thus, we should increase the size of T1 for which the value of *p* should increase. If there is a hit in B2 then the data stored from the point of view of frequency was more relevant and more space should be allotted to T2. Thus, the value of *p* should decrease. The amount by which *p* should deviate is given by the relative sizes of B1 and B2.

E. CLOCK with Adaptive Replacement (CAR)

CAR attempts to merge the adaptive policy of ARC with the implementation efficiency of CLOCK [5]. The algorithm maintains four doubly linked lists T1, T2, B1, and B2. T1 and T2 are CLOCKs while B1 and B2 are simple LRU lists. The concept behind these lists is same as that for ARC. In addition, the lists T1 and T2 i.e. the pages in the cache, have a reference bit that can be set or reset.
The precise definition of four lists is as follows:
1. *T*10 and *B*1 contains all the pages that are referenced exactly once since its most recent eviction from *T*1 U *T*2 U *B*1 U *B*2 or was never referenced before since its inception.
2. *T*11 , *B*2 and *T*2 contains all the pages that are referenced
more than once since its most recent eviction from *T*1 U *T*2 U *B*1 U *B*2.
The two important constraints on the sizes of T1, T2, B1and B2 are:
1. $0 \leq |T1|+|B1| \leq c$. By definition, *T*1 U *B*1 captures recency. The size of recently accessed pages and frequently accessed pages keep on changing. This prevents pages which are accessed only once from taking up the entire cache directory of size 2*c* since increasing size of *T*1 U *B*1 indicates that the recently referenced pages are not being referenced again which in turn means the recency data that is stored is not helpful. Thus it means that only the frequently used pages are re-referenced or new pages are being referenced.
2. $0 \leq |T2|+|B2| \leq 2c$. If only a set of pages are being accessed frequently, there are no new references. The cache directory has information regarding only frequency.

F. Aging

The aging algorithm is somewhat tricky: It uses a bit field of w bits for each page in order to track its accessing profile. Every time a page is read, the first (i.e. most significant) bit of the page's bit field is set. Every n instructions all pages' bit fields are right-shifted by one bit. The next page to replace is the one with the lowest (numerical) value of its bit field. If there are several pages having the same value, an arbitrary page is chosen. The aging algorithm works very well in many cases, and sometimes even better than LRU, because it looks behind the last access. It furthermore is rather easy to implement, because there are no expensive actions to perform when reading a page. However, finding the page with the lowest bit field value usually takes some time. Thus, it might be necessary to predetermine the next page to be swapped out in background [6].

ANALYSIS

Offline performance of the algorithms is measured as page fault count and hit ratio.
Hit ratio (hr) is calculated as

hr = 100 - mr.

Miss ratio (mr) is

mr = 100 _ ((#pf - #distinct)/(#refs - #distinct)),

where #pf is the number of page faults, #distinct is the number of distinct pages used in the trace and #refs is The number of references in the trace.

CONCLUSION

The evolution of replacement algorithms shows the analyses and proof of better performance has moved from mathematical analysis to testing against real world program traces. This trend shows how difficult it is to mathematically model the memory behavior of programs. An important factor is also the large amount and easy availability of important programs. The other clear trend is the realization of the need for workload adaption. The simple traces used in this thesis support the conclusions of the authors. CAR and ARC seem most promising algorithms and offer significant improvement over basic CLOCK. Page replacement plays only a small part in overall performance of applications, but studies, have shown that the benefits are real. It certainly seems like a worthwhile idea to further evaluate implementations of both CAR and ARC in real operating system.

REFERENCES:

[1] A. S. Sumant, and P. M. Chawan, "Virtual Memory Management Techniques in 2.6 Linux kernel and challenges", IASCIT International Journal of Engineering and Technology, pp. 157-160, 2010.
[2] Heikki Paajanen, Page replacement in operating system memory Management, Master's thesis, University of Jyväskylä, 2007
[3] Amit S. Chavan, Kartik R. Nayak, Keval D. Vora, Manish D. Purohit and Pramila M. Chawan, A Comparison of Page Replacement Algorithms, IACSIT, vol.3, no.2, April 2011.
[4] N. Meigiddo, and D. S. Modha, "ARC: A Self-Tuning, Low overhead Replacement Cache", *IEEE Transactions on Computers*, pp. 58-65, 2004.
[5] S. Bansal, and D. Modha, "CAR: Clock with Adaptive Replacement", *FAST-'04 Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pp. 187-200, 2004.
Mohd Zeeshan Farooqui, Mohd Shoaib, Mohammad unnun Khan, A Comprehensive Survey of Page Replacement Algorithms, IJARCET, VOLUME 3 ISSUE 1, JANUARY 2014