S.M. Shamsheer Daula, Dr. K.E Sreenivasa Murthy, G Amjad Khan / International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622
www.ijera.com Vol. 2, Issue 2, Mar-Apr 2012, pp.126-130

A Throughput Analysis on Page Replacement Algorithms in Cache Memory Management

S.M. Shamsheer Daula*, Dr. K.E Sreenivasa Murthy, G Amjad Khan*****

*Asst. Professor, G. Pulla Reddy Engineering College, Kurnool, A.P India
** Principal, SKTRMC, Kondair, Andhra Pradesh, India
*** Asst. Professor, G. Pulla Reddy Engineering College, Kurnool, A. P, India

Abstract: The less time any processor takes in handling instructions the more would be its speed and efficiency. The speed of a processing device is not only based on it's Architectural features or Operational frequencies but also it needs to be dependent on the Memory mapping. A processors memory management is helped out with the operating system on which it is working. The memory management subsystem is one of the most important parts of the operating system. Since the early days of computing, there has been a need for more memory than exists physically in a system. Many considerations were raised leading to concepts of virtual and physical memories. With all steps on improving side a concept of Cache Memory is also much impressive. The nearest block of storage to hike processor speed has the procedures of fast information transactions. The retrieval of information (pages) from memories could be brought to fast extent by fast replacements of pages. This paper presents the various replacement algorithms with their performance analysis.

General Terms: Memories, Replacement Algorithms, Memory Mapping, Cache , Virtual Memory.

Keywords: LRU, LFU, FIFO, ARC, CAR, Linux, Hit, Miss.

1. Introduction

The memory management subsystem is one of the most important parts of the operating system. In the Hierarchy status of memory the Cache memory stands at the nearest level to transact information towards the processing unit. Since the early days of computing, there has been a need for more memory than exists physically in a system. Apart from making the processors, memory[3]so on faster the best approach is to maintain caches of useful information and data that make some operations faster. Linux uses a number of memory management related caches.

Strategies have been developed to overcome this limitation and the most successful of these is virtual memory. Virtual memory makes the system appear to have more memory than it actually has by sharing it between competing processes as they need it. In most operating system texts, the treatment of memory management includes a section entitled "replacement policy," which deals with the selection of a page in main memory to be replaced when a new page must be brought in. This topic is sometimes difficult to explain because several interrelated concepts are involved:

-- How many page frames are to be allocated to each active process
-- Whether the set of pages to be considered for replacement should be limited to those of the process that caused the page fault or encompass all the page frames in main memory
-- Among the set of pages considered, which particular page should be selected for replacement

2. The Memory Management Subsystem:

Virtual memory does more than just make your computer's memory go further. The memory management subsystem provides:

Large Address Spaces :The operating system makes the system appear as if it has a larger amount of memory than it actually has. The virtual memory can be many times larger than the physical memory in the system.[1,12]

Protection:Each process in the system has its own virtual address space. These virtual address spaces are completely separate from each other and so a process running one application cannot affect another. Also, the hardware virtual memory mechanisms allow areas of memory to be protected against writing. This protects code and data from being overwritten by rogue applications.

Memory Mapping :Memory mapping is used to map image and data files into a processes address space. In

memory mapping, the contents of a file are linked directly into the virtual address space of a process.

Fair Physical Memory Allocation :The memory management subsystem allows each running process in the system a fair share of the physical memory of the system.

Shared Virtual Memory :Although virtual memory allows processes to have separate (virtual) address spaces, there are times when you need processes to share memory. For example there could be several processes in the system running the bash command shell. Rather than have several copies of bash, one in each processes virtual address space, it is better to have only one copy in physical memory and all of the processes running bash share it. Dynamic libraries are another common example of executing code shared between several processes. Shared memory can also be used as an Inter Process Communication (IPC) [22,8,17] mechanism, with two or more processes exchanging information via memory common to all of them. Linux supports the Unix [TM] System V shared memory IPC.

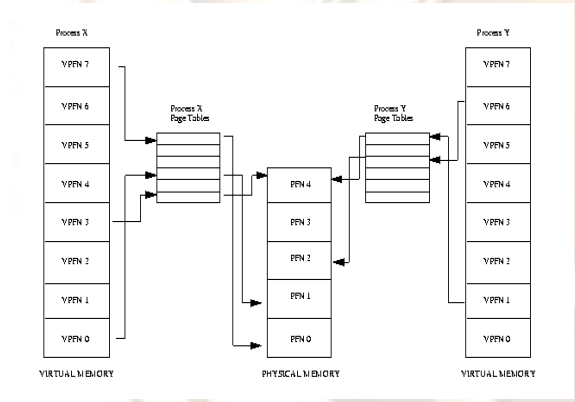

Figure 1: Abstract model of Virtual to Physical address mapping

3. Cache Memory:

Apart from making the processors, memory and so on faster the best approach is to maintain caches of useful information and data that make some operations faster. Linux uses a number of memory management related caches:

Buffer Cache:The buffer cache contains data buffers that are used by the block device drivers.
These buffers are of fixed sizes (for example 512 bytes) and contain blocks of information that have either been read from a block device or are being written to it. A block device is one that can only be accessed by reading and writing fixed sized blocks of data. All hard disks are block devices.
The buffer cache is indexed via the device identifier and the desired block number and is used to quickly find a block of data. Block devices are only ever accessed via the buffer cache. If data can be found in the buffer cache then it does not need to be read from the physical block device, for example a hard disk, and access to it is much faster.

Page Cache : This is used to speed up access to images and data on disk. It is used to cache the logical contents of a file a page at a time and is accessed via the file and offset within the file. As pages are read into memory from disk, they are cached in the page cache.

Swap Cache :Only modified (or *dirty*) pages are saved in the swap file. So long as these pages are not modified after they have been written to the swap file then the next time the page is swapped out there is no need to write it to the swap file as the page is already in the swap file. Instead the page can simply be discarded. In a heavily swapping system this saves many unnecessary and costly disk operations.[11,16,20]

Hardware Caches: One commonly implemented hardware cache is in the processor; a cache of Page Table Entries. In this case, the processor does not always read the page table directly but instead caches translations for pages as it needs them. These are the Translation Look-aside Buffers and contain cached copies of the page table entries from one or more processes in the system. When the reference to the virtual address is made, the processor will attempt to find a matching TLB entry. If it finds one, it can directly translate the virtual address into a physical one and perform the correct operation on the data. If the processor cannot find a matching TLB entry then it must get the operating system to help. It does this by signalling the operating system that a TLB miss has occurred. A system specific mechanism is used to deliver that exception to the operating system code that can fix things up. The operating system generates a new TLB entry for the address mapping. When the exception has been cleared, the processor will make another attempt to translate the virtual address. This time it will work because there is now a valid entry in the TLB for that address.

4. Page Replacement Algorithms :

A virtual memory system needs efficient page replacement algorithms to decide which pages to evict from memory in case of a page fault.[6,5,13,14,15] Over the years many algorithms have been proposed

for page replacement. Each algorithm attempts 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. A traditional CPU implementation, that contribute to the increase of code density and faster execution.

Both cache and auxiliary memory handle uni-formly sized items called pages. Requests for pages go first to the cache. When a page is found in the cache, a hit occurs; otherwise, a cache miss hap-pens, and the request goes to the auxiliary memory. In the latter case, a copy is paged in to the cache. This practice, called demand paging, rules out prefetching pages from the auxiliary memory into the cache. If the cache is full, before the system can page in a new page, it must page out one of the cur-rently cached pages.[7] A replacement policy deter-mines which page is evicted.

4.1 Optimal and FIFO Algortihms:

The optimal policy selects for replacement that page for which the time to the next reference is the longest. It can be shown that this policy results in the fewest number of page faults.[1,16] Clearly, this policy is impossible to implement, because it would require the operating system to have perfect knowledge of future events. However, it does serve as a standard against which to judge real-world algorithms.

The first-in-first-out (FIFO) policy treats the page frames allocated to a process as a circular buffer, and pages are removed in round-robin style. All that is required is a pointer that circles through the page frames of the process. This is therefore one of the simplest page replacement policies to implement. The logic behind this choice, other than its simplicity, is that one is replacing the page that has been in memory the longest: A page fetched into memory a long time ago may have now fallen out of use. This reasoning will often be wrong, because there will often be regions of program or data that are heavily used throughout the life of a program. Those pages will be repeatedly paged in and out by the FIFO algorithm

4.2 LRU:

A commonly used criterion for evaluating a replacement policy is its hit ratio—the frequency with which it finds a page in the cache. Of course, the replacement policy's implementation overhead should not exceed the anticipated time savings. Discarding the least-recently-used page is the pol-icy of choice in cache management.

Until recently, attempts to outperform LRU in practice had notsucceeded because of overhead issues and the need to pretune parameters. The adaptive replacement cache is a self-tuning, low-overhead algorithm that responds online to changing access patterns. ARC continually balances between the recency and fre-quency features of the workload, demonstrating that adaptation eliminates the need for the work-load-specific pretuning that plagued many previous proposals to improve LRU.[5,6,14] ARC's online adaptation will likely have benefits for real-life workloads due to their richness and variability with time. These workloads can contain long sequential I/Os or moving hot spots, changing frequency and scale of temporal locality and fluc-tuating between stable, repeating access patterns and patterns with transient clustered references.

4.2 ARC[Adaptive Replacement Cache]:

Like LRU, ARC is easy to implement, and its run-ning time per request is essentially independent of the cache size. A real-life implementation revealed that ARC has a low space overhead—0.75 percent of the cache size. Also, unlike LRU, ARC is scan-resistant in that it allows one-time sequential requests to pass through without polluting the cache or flushing pages that have temporal locality.[17,11].

Likewise, ARC also effectively handles long peri-ods of low temporal locality. ARC leads to sub-The self-tuning, low-overhead, scan-resistant adaptive replacement cache algorithm outperforms the least-recently-used algorithm by dynamically responding to changing access patterns and continually balancing between workload recency and frequency features. stantial performance gains in terms of an improved hit ratio compared with LRU for a wide range of cache sizes.[6]

4.3 CAR [CLOCK with Adaptive Replacement]:

CLOCK is a classical cache replacement policy dating back to 1968 that was proposed as a low-complexity approximation to LRU. On every cache hit, the policy LRU needs to move the accessed item to the most recently used position, at which point, to ensure consistency and correctness, it serializes cache hits behind a single global lock.[23,15,16]inates this lock contention. The algorithm CAR is inspired by the Adaptive Replacement Cache (ARC) algorithm, and inherits virtually all advantages of ARC including its high performance, but does not serialize cache hits behind a single global lock.

The average memory reference time is

$$T = m * Tm + Th + E$$

CAR is a refinement of the Clock algorithm using the principles of ARC . The basic idea is to maintain two clocks, say, T1 and T2, where T1 contains pages with "recency" or "short-term utility" and T2 contains pages with "frequency" or "longterm utility". New pages ar first inserted in T1 and graduate to T2 upon passing a certain test of long-term utility.[8,17,19] By using a certain precise history mechanism that remembers recently evicted pages from T1 and T2, algorithm adaptively determine the sizes of these lists in a data-driven fashion. CAR has a very low overhead on cache hits.

CAR is self-tuning. The policy CAR requires no Tunable parameters, it is scan-resistant. A scan is any sequence of one-time use requests. CAR has very low space over head.

4.3.1 Related Operation:

The first key point of the above algorithm is the simplicity, where cache hits are not serialized behind a lock and virtually no overhead is involved. [22,10,17,18]. The second , key point is the continual adaptation of the target size. This observation produced by the below presented steps of operation(Part of the complete function is presented)

CAR(x)
INPUT: The requested page x.
if (x is in T1 ∪ T2) then /* cache hit */
Set the page reference bit for x to one.
else /* cache miss */
if (| T1 | + | T2 | = c) then
/* cache full, replace a page from cache */
replace( ) /* cache directory replacement */
if ((x is not in B 1 ∪ B2) and (| T1 | + | B 1 | = c)) then
Discard the LRU page in B 1 .
elseif ((| T1 | + | T2 | + | B 1 | + | B 2 | = 2 c) and (x is not in B 1 ∪ B2)) then
Discard the LRU page in B 2 .
endif
endif /* cache directory miss */
if (x is not in B 1 ∪ B2) then
Insert x at the tail of T1 . Set the page reference bit of x to 0. /* cache directory hit */
elseif (x is in B 1) then
Adapt: Increase the target size for the list T1 as: p = min {p + max {1, | B
Move x at the tail of T2 . Set the page reference bit of x to 0. /* cache directory hit */
else /* x must be in B 2 */
Adapt: Decrease the target size for the list T1 as: p = max {p − max {1, |
Move x at the tail of T2 . Set the page reference bit of x to 0.
endif
endif

replace()
found = 0
repeat
if (| T1 | >= max(1 , p)) then
if (the page reference bit of head page in T 1 is 0) then
found = 1;
Demote the head page in T1 and make it the MRU page in B 1 .
else
Set the page reference bit of head page in T 1 to 0, and make it the ta
endif
else
if (the page reference bit of head page in T 2 is 0), then
found = 1;
Demote the head page in T2 and make it the MRU page in B 2 .
else
Set the page reference bit of head page in T 2 to 0, and make it the ta
endif
endif
until (found)

5. Experimental Results:

A case of procedure of each explained algorithm is figured by taking page request with page size of 4 KB to 4 MB over a memory block of 16 GB, The results are traced based on the percentage values, the high level end embedded C source is coded to get the results, The Comparisons are Tabulated as

Cache Pages	Percentage of Hit Ratios				
	CAR	ARC	LRU	FIFO	OPT
1,024	42.43	39.17	38.41	33.8	32.27
2,048	51.68	42.2	41.44	36.2	36.12
4,096	55.4	47.59	46.05	39.13	38.24
8,192	65.13	56.14	53.06	44.8	44.19

Table 1: A comparison of Hit Ratios of the Replacement algorithms

6. Conclusions and Future works:

CAR removes the cache hit serialization problem of LRU and ARC. The CAR attempts to merge the adaptive policy of ARC with the implementation efficiency of CLOCK. The self-tuning nature of CAR makes it very attractive for deployment in environments where no a priori knowledge of the workloads is available. CAR is scan-resistant. A scan is any sequence ofone-time use requests.

CAR moves ahead with new approach CART- Clock Adaptive Replacement algorithm with Temporal

filtering, which has all the features and advantages of CAR in addition, it employs a much stricter and more precise criterion to distinguish pages with short-term utility from those with long-term utility.

6. References

[1] L. A. Belady, "A study of replacement algorithms for virtual storage computers," IBM Sys. J., vol. 5, no. 2, pp. 78–101, 1966.

[2] M. J. Bach, The Design of the UNIX Operating System. Engle- wood Cliffs, NJ: Prentice-Hall, 1986.

[3] A. S. Tanenbaum and A. S. Woodhull, Operating Systems:
Design and Implementation. Prentice-Hall, 1997.

[4] A. Silberschatz and P. B. Galvin, Operating System Concepts.Reading, MA: Addison-Wesley, 1995.

[5] J. E. G. Coffman and P. J. Denning, Operating Systems Theory. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[6] F. J. Corbat ´o, "A paging experiment with the multics system," in In Honor of P. M. Morse, pp. 217–228, MIT Press, 1969. Also as MIT Project MAC Report MAC-M-384, May 1968.

[7] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F.
Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and
E. Shekita, "Starburst mid-flight: As the dust clears," IEEE
Trans. Knowledge and Data Engineering, vol. 2, no. 1, pp. 143– 160, 1990.

[8] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarter-
man, The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley, 1996..

[9] 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.

[10]A. Janapsatya, A. Ignjatovic, J. Peddersen and S. Parameswaran, "Dueling CLOCK: Adaptive cache replacement policy based on the CLOCK algorithm", Design, Automation and Test in Europe Conference and Exhibition, pp. 920-925, 2010.

[11] S. Jiang, and X. Zhang, "LIRS: An Efficient Policy to improve Buffer Cache Performance", IEEE Transcations on Computers, pp. 939-952, 2005.

[12] S. Jiang, X. Zhang, and F. Chen, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement", ATEC '05 Proceedings of the annual conference on USENIX Anuual Tecchnical Conference, pp. 35, 2005.

[13] N. Meigiddo, and D. S. Modha, "ARC: A Self-Tuning, Low overhead Replacement Cache", IEEE Transactions on Computers, pp. 58-65, 2004.

[14] J. E. O'neil, P. E. O'neil and G. Weikum, "An optimality Proof of the LRU-K Page Replacement Algorithm", Journal of the ACM, pp. 92-112, 1999.

[15] 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.

[16] D. Lee et al., "LRFU: A Spectrum of Policies that Sub-sumes the Least Recently Used and Least Frequently
Used Policies," IEEE Trans. Computers, vol. 50, no.2, 2010, pp. 1352-1360.

[17] Y. Zhou and J.F. Philbin, "The Multi-Queue Replace-ment Algorithm for Second-Level Buffer Caches," Proc. Usenix Ann. Tech. Conf. (Usenix 2010), Usenix, 2010, pp. 91-104.

[18] W.W. Hsu, A.J. Smith, and H.C. Young, The Auto-matic Improvement of Locality in Storage Systems, tech. report, Computer Science Division, Univ. of California, Berkeley, 2010.

[19] N. Megiddo and D.S. Modha, "ARC: A Self-Tuning,
Low Overhead Replacement Cache," Proc. Usenix Conf. File and Storage Technologies (FAST 2009),
Usenix, 2010, pp. 115-130.

[20] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min,
Y. Cho, and C. S. Kim, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," IEEE Trans. Computers, vol. 50, no. 12, pp. 1352–1360, 2009.

[21] Y. Zhou and J. F. Philbin, "The multi-queue replacement algo-rithm for second level buffer caches," in Proc. USENIX Annual Tech. Conf. (USENIX 2001), Boston, MA, pp. 91–104, June 2010.

[22] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to
improve buffer cache perfor-mance," in Proc. ACM SIGMETRICS Conf., 2009