Page replacement algorithm

From Wikipedia, the free encyclopedia
(Redirected from Page replacement algorithms)

This article is about algorithms specific to paging. For outline of general cache algorithms (e.g. processor, disk, database, web), see Cache algorithms.

In a computer operating system that uses paging for virtual memory management, **page replacement algorithms** decide which memory pages to page out (swap out, write to disk) when a page of memory needs to be allocated. Paging happens when a page fault occurs and a free page cannot be used to satisfy the allocation, either because there are none, or because the number of free pages is lower than some threshold.

When the page that was selected for replacement and paged out is referenced again it has to be paged in (read in from disk), and this involves waiting for I/O completion. This determines the *quality* of the page replacement algorithm: the less time waiting for page-ins, the better the algorithm. A page replacement algorithm looks at the limited information about accesses to the pages provided by hardware, and tries to guess which pages should be replaced to minimize the total number of page misses, while balancing this with the costs (primary storage and processor time) of the algorithm itself.

The page replacing problem is a typical online problem from the competitive analysis perspective in the sense that the optimal deterministic algorithm is known.

History [edit]

Page replacement algorithms were a hot topic of research and debate in the 1960s and 1970s. That mostly ended with the development of sophisticated LRU (least recently used) approximations and working set algorithms. Since then, some basic assumptions made by the traditional page replacement algorithms were invalidated, resulting in a revival of research. In particular, the following trends in the behavior of underlying hardware and user-level software have affected the performance of page replacement algorithms:

- Size of primary storage has increased by multiple orders of magnitude. With several gigabytes of primary memory, algorithms that require a periodic check of each and every memory frame are becoming less and less practical.

- Memory hierarchies have grown taller. The cost of a CPU cache miss is far more expensive. This exacerbates the previous problem.
- Locality of reference of user software has weakened. This is mostly attributed to the spread of object-oriented programming techniques that favor large numbers of small functions, use of sophisticated data structures like trees and hash tables that tend to result in chaotic memory reference patterns, and the advent of garbage collection that drastically changed memory access behavior of applications.

Requirements for page replacement algorithms have changed due to differences in operating system kernel architectures. In particular, most modern OS kernels have unified virtual memory and file system caches, requiring the page replacement algorithm to select a page from among the pages of both user program virtual address spaces and cached files. The latter pages have specific properties. For example, they can be locked, or can have write ordering requirements imposed by journaling. Moreover, as the goal of page replacement is to minimize total time waiting for memory, it has to take into account memory requirements imposed by other kernel sub-systems that allocate memory. As a result, page replacement in modern kernels (Linux, FreeBSD, and Solaris) tends to work at the level of a general purpose kernel memory allocator, rather than at the higher level of a virtual memory subsystem.

Local vs. global replacement [edit]

Replacement algorithms can be *local* or *global*.

When a process incurs a page fault, a local page replacement algorithm selects for replacement some page that belongs to that same process (or a group of processes sharing a memory partition). A global replacement algorithm is free to select any page in memory.

Local page replacement assumes some form of memory partitioning that determines how many pages are to be assigned to a given process or a group of processes. Most popular forms of partitioning are *fixed partitioning* and *balanced set* algorithms based on the working set model. The advantage of local page replacement is its scalability: each process can handle its page faults independently without contending for some shared global data structure.

Precleaning [edit]

Most replacement algorithms simply return the target page as their result. This means that if target page is *dirty* (that is, contains data that have to be written to the stable storage before page can be reclaimed), I/O has to be initiated to send that page to the stable storage (to *clean* the page). In the early days of virtual memory, time spent on cleaning was not of much concern, because virtual memory was first implemented on systems with full duplex channels to the stable storage, and cleaning was customarily overlapped with paging. Contemporary commodity hardware, on the other hand, does not support full duplex transfers, and cleaning of target pages becomes an issue.

To deal with this situation, various *precleaning* policies are implemented. Precleaning is the mechanism that starts I/O on dirty pages that are (likely) to be replaced soon. The idea is that by the time the precleaned page is actually selected for the replacement, the I/O will complete and the page will be clean. Precleaning assumes that it is possible to identify pages that will be replaced *next*. Precleaning that is too eager can waste I/O bandwidth by writing pages that manage to get re-dirtied before being selected for replacement.

Anticipatory paging [edit]

See also: Paging

Some systems use demand paging—waiting until a page is actually requested before loading it into RAM.

Other systems attempt to reduce latency by guessing which pages not in RAM are likely to be needed soon, and pre-loading such pages into RAM, before that page is requested. (This is often in combination with pre-cleaning, which guesses which pages currently in RAM are not likely to be needed soon, and pre-writing them out to storage).

When a page fault occurs, "anticipatory paging" systems will not only bring in the referenced page, but also the next few consecutive pages (analogous to a prefetch input queue in a CPU).

The swap prefetch mechanism goes even further in loading pages (even if they are not consecutive) that are likely to be needed soon.

This article **may be confusing or unclear to readers**. Please help us clarify the article; suggestions may be found on the talk page. *(January*

The (h,k)-paging problem [edit]

The (h,k)-paging problem is a generalization of the model of paging problem: Let h,k be positive integers that $h \leq k$. We measure the performance of an algorithm with cache of size $h \leq k$ relative to the theoretically optimal page replacement algorithm. If $h < k$ we provide the optimal page replacement algorithm with strictly less resource.

The (h,k)-paging problem is a way to measure how an online algorithm performs by comparing it with the performance of the optimal algorithm, specifically, separately parameterizing the cache size of the online algorithm and optimal algorithm.

Marking algorithms [edit]

Marking algorithms is a general class of paging algorithms. For each page, we associate it with a bit called its mark. Initially, we set all pages as unmarked. During a stage of page requests, we mark a page when it is first requested in this stage. A marking algorithm is such an algorithm that never pages out a marked page.

If ALG is a marking algorithm with a cache of size k, and OPT is the optimal algorithm with a cache of $h \leq k$, then ALG is $\dfrac{k}{k-h+1}$-competitive. So every marking algorithm attains the $\dfrac{k}{k-h+1}$-competitive ratio.

LRU is a marking algorithm while FIFO is not a marking algorithm.

Conservative algorithms [edit]

An algorithm is conservative, if on any consecutive request sequence containing k or fewer distinct page references, the algorithm will incur k or fewer page faults.

If ALG is a conservative algorithm with a cache of size k, and OPT is the optimal algorithm with a cache of $h \leq k$, then ALG is $\dfrac{k}{k-h+1}$-competitive. So every conservative algorithm attains the $\dfrac{k}{k-h+1}$-competitive ratio.

LRU, FIFO and CLOCK are conservative algorithms.

Page replacement algorithms [edit]

There are a variety of page replacement algorithms:[1]

The theoretically optimal page replacement algorithm [edit]

The theoretically optimal page replacement algorithm (also known as OPT, clairvoyant replacement algorithm, or Bélády's optimal page replacement policy)[2][3][4] is an algorithm that works as follows: when a page needs to be swapped in, the operating system swaps out the page whose next use will occur farthest in the future. For example, a page that is not going to be used for the next 6 seconds will be swapped out over a page that is going to be used within the next 0.4 seconds.

This algorithm cannot be implemented in a general purpose operating system because it is impossible to compute reliably how long it will be before a page is going to be used, except when all software that will run on a system is either known beforehand and is amenable to static analysis of its memory reference patterns, or only a class of applications allowing run-time analysis. Despite this limitation, algorithms exist[*citation needed*] that can offer near-optimal performance — the operating system keeps track of all pages referenced by the program, and it uses those data to decide which pages to swap in and out on subsequent runs. This algorithm can offer near-optimal performance, but not on the first run of a program, and only if the program's memory reference pattern is relatively consistent each time it runs.

Analysis of the paging problem has also been done in the field of online algorithms. Efficiency of randomized online algorithms for the paging problem is measured using amortized analysis.

Not recently used [edit]

The not recently used (NRU) page replacement algorithm is an algorithm that favours keeping pages in memory that have been recently used. This algorithm works on the following principle: when a page is referenced, a referenced bit is set for that page, marking it as referenced. Similarly, when a page is modified (written to), a modified bit is set. The setting of the bits is usually done by the hardware, although it is possible to do so on the

software level as well.

At a certain fixed time interval, the clock interrupt triggers and clears the referenced bit of all the pages, so only pages referenced within the current clock interval are marked with a referenced bit. When a page needs to be replaced, the [operating system](#) divides the pages into four classes:

3. referenced, modified
2. referenced, not modified
1. not referenced, modified
0. not referenced, not modified

Although it does not seem possible for a page to be not referenced yet modified, this happens when a class 3 page has its referenced bit cleared by the clock interrupt. The NRU algorithm picks a random page from the lowest category for removal. So out of the above four pages, the NRU algorithm will replace the not referenced, not modified. Note that this algorithm implies that a modified but not referenced (within last clock interval) page is less important than a not modified page that is intensely referenced.

NRU is a marking algorithm, so it is $\dfrac{k}{k-h+1}$-competitive.

First-in, first-out [edit]

The simplest page-replacement algorithm is a FIFO algorithm. The first-in, first-out (FIFO) page replacement algorithm is a low-overhead algorithm that requires little bookkeeping on the part of the [operating system](#). The idea is obvious from the name – the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the oldest arrival in front. When a page needs to be replaced, the page at the front of the queue (the oldest page) is selected. While FIFO is cheap and intuitive, it performs poorly in practical application. Thus, it is rarely used in its unmodified form. This algorithm experiences [Bélády's anomaly](#).

FIFO page replacement algorithm is used by the [VAX/VMS](#) operating system, with some modifications.[5] Partial second chance is provided by skipping a limited number of entries with valid translation table references,[6] and additionally, pages are displaced from process working set to a systemwide pool from which they can be recovered if not already re-used.

FIFO is a conservative algorithm, so it is $\dfrac{k}{k-h+1}$-competitive.

Second-chance [edit]

A modified form of the FIFO page replacement algorithm, known as the Second-chance page replacement algorithm, fares relatively better than FIFO at little cost for the improvement. It works by looking at the front of the queue as FIFO does, but instead of immediately paging out that page, it checks to see if its referenced bit is set. If it is not set, the page is swapped out. Otherwise, the referenced bit is cleared, the page is inserted at the back of the queue (as if it were a new page) and this process is repeated. This can also be thought of as a circular queue. If all the pages have their referenced bit set, on the second encounter of the first page in the list, that page will be swapped out, as it now has its referenced bit cleared. If all the pages have their reference bit set then second chance algorithm degenerates into pure FIFO.

As its name suggests, Second-chance gives every page a "second-chance" – an old page that has been referenced is probably in use, and should not be swapped out over a new page that has not been referenced.

Clock [edit]

Clock is a more efficient version of FIFO than Second-chance because pages don't have to be constantly pushed to the back of the list, but it performs the same general function as Second-Chance. The clock algorithm keeps a circular list of pages in memory, with the "hand" (iterator) pointing to the last examined page frame in the list. When a page fault occurs and no empty frames exist, then the R (referenced) bit is inspected at the hand's location. If R is 0, the new page is put in place of the page the "hand" points to, otherwise the R bit is cleared. Then, the clock hand is incremented and the process is repeated until a page is replaced.[7]

Variants of Clock [edit]

- GCLOCK: Generalized clock page replacement algorithm.[8]
- Clock-Pro keeps a circular list of information about recently referenced pages, including all M pages in memory as well as the most recent M pages that have been paged out. This extra information on paged-out pages, like the similar information maintained by [ARC](#), helps it work better than LRU on large loops and one-time scans.[9]

- WSclock.[10] The "aging" algorithm and the "WSClock" algorithm are probably the most important page replacement algorithms in practice.[11][12]
- Clock with Adaptive Replacement (CAR) is a page replacement algorithm that has performance comparable to ARC, and substantially outperforms both LRU and CLOCK.[13] The algorithm CAR is self-tuning and requires no user-specified magic parameters.

CLOCK is a conservative algorithm, so it is $\dfrac{k}{k-h+1}$-competitive.

Least recently used [edit]

The least recently used (LRU) page replacement algorithm, though similar in name to NRU, differs in the fact that LRU keeps track of page usage over a short period of time, while NRU just looks at the usage in the last clock interval. LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too. While LRU can provide near-optimal performance in theory (almost as good as Adaptive Replacement Cache), it is rather expensive to implement in practice. There are a few implementation methods for this algorithm that try to reduce the cost yet keep as much of the performance as possible.

The most expensive method is the linked list method, which uses a linked list containing all the pages in memory. At the back of this list is the least recently used page, and at the front is the most recently used page. The cost of this implementation lies in the fact that items in the list will have to be moved about every memory reference, which is a very time-consuming process.

Another method that requires hardware support is as follows: suppose the hardware has a 64-bit counter that is incremented at every instruction. Whenever a page is accessed, it acquires the value equal to the counter at the time of page access. Whenever a page needs to be replaced, the operating system selects the page with the lowest counter and swaps it out. With present hardware, this is not feasible because the OS needs to examine the counter for every page in the cache memory.

Because of implementation costs, one may consider algorithms (like those that follow) that are similar to LRU, but which offer cheaper implementations.

One important advantage of the LRU algorithm is that it is amenable to full statistical analysis. It has been proven, for example, that LRU can never result in more than N-times more page faults than OPT algorithm, where N is proportional to the number of pages in the managed pool.

On the other hand, LRU's weakness is that its performance tends to degenerate under many quite common reference patterns. For example, if there are N pages in the LRU pool, an application executing a loop over array of N + 1 pages will cause a page fault on each and every access. As loops over large arrays are common, much effort has been put into modifying LRU to work better in such situations. Many of the proposed LRU modifications try to detect looping reference patterns and to switch into suitable replacement algorithm, like Most Recently Used (MRU).

Variants on LRU [edit]

1. LRU-K 📄 evicts the page whose K-th most recent access is furthest in the past. For example, LRU-1 is simply LRU whereas LRU-2 evicts pages according to the time of their penultimate access. LRU-K improves greatly on LRU with regards to locality in time.
2. The ARC[14] algorithm extends LRU by maintaining a history of recently evicted pages and uses this to change preference to recent or frequent access. It is particularly resistant to sequential scans.

A comparison of ARC with other algorithms (LRU, MQ, 2Q, LRU-2, LRFU, LIRS) can be found in Megiddo & Modha.[15]

LRU is a marking algorithm, so it is $\dfrac{k}{k-h+1}$-competitive.

Random [edit]

Random replacement algorithm replaces a random page in memory. This eliminates the overhead cost of tracking page references. Usually it fares better than FIFO, and for looping memory references it is better than LRU, although generally LRU performs better in practice. OS/390 uses global LRU approximation and falls back to random replacement when LRU performance degenerates, and the Intel i860 processor used a random replacement policy (Rhodehamel 1989).

Not frequently used [edit]

The not frequently used (NFU) page replacement algorithm requires a counter, and every page has one counter of its own which is initially set to 0. At each clock interval, all pages that have been referenced within that interval will have their counter incremented by 1. In effect, the counters keep track of how frequently a page has been used. Thus, the page with the lowest counter can be swapped out when necessary.

The main problem with NFU is that it keeps track of the frequency of use without regard to the time span of use. Thus, in a multi-pass compiler, pages which were heavily used during the first pass, but are not needed in the second pass will be favoured over pages which are comparably lightly used in the second pass, as they have higher frequency counters. This results in poor performance. Other common scenarios exist where NFU will perform similarly, such as an OS boot-up. Thankfully, a similar and better algorithm exists, and its description follows.

The not frequently used page-replacement algorithm generates fewer page faults than the least recently used page replacement algorithm when the page table contains null pointer values.

Aging [edit]

The aging algorithm is a descendant of the NFU algorithm, with modifications to make it aware of the time span of use. Instead of just incrementing the counters of pages referenced, putting equal emphasis on page references regardless of the time, the reference counter on a page is first shifted right (divided by 2), before adding the referenced bit to the left of that binary number. For instance, if a page has referenced bits 1,0,0,1,1,0 in the past 6 clock ticks, its referenced counter will look like this: 10000000, 01000000, 00100000, 10010000, 11001000, 01100100. Page references closer to the present time have more impact than page references long ago. This ensures that pages referenced more recently, though less frequently referenced, will have higher priority over pages more frequently referenced in the past. Thus, when a page needs to be swapped out, the page with the lowest counter will be chosen.

Note that aging differs from LRU in the sense that aging can only keep track of the references in the latest 16/32 (depending on the bit size of the processor's integers) time intervals. Consequently, two pages may have referenced counters of 00000000, even though one page was referenced 9 intervals ago and the other 1000 intervals ago. Generally speaking, knowing the usage within the past 16 intervals is sufficient for making a good decision as to which page to swap out. Thus, aging can offer near-optimal performance for a moderate price.

Techniques for hardware with no reference bit [edit]

Many of the techniques discussed above assume the presence of a reference bit associated with each page. Some hardware has no such bit, so its efficient use requires techniques that operate well without one.

One notable example is VAX hardware running OpenVMS. This system knows if a page has been modified, but not necessarily if a page has been read. Its approach is known as Secondary Page Caching. Pages removed from working sets (process-private memory, generally) are placed on special-purpose lists while remaining in physical memory for some time. Removing a page from a working set is not technically a page-replacement operation, but effectively identifies that page as a candidate. A page whose backing store is still valid (whose contents are not dirty, or otherwise do not need to be preserved) is placed on the tail of the Free Page List. A page that requires writing to backing store will be placed on the Modified Page List. These actions are typically triggered when the size of the Free Page List falls below an adjustable threshold.

Pages may be selected for working set removal in an essentially random fashion, with the expectation that if a poor choice is made, a future reference may retrieve that page from the Free or Modified list before it is removed from physical memory. A page referenced this way will be removed from the Free or Modified list and placed back into a process working set. The Modified Page List additionally provides an opportunity to write pages out to backing store in groups of more than one page, increasing efficiency. These pages can then be placed on the Free Page List. The sequence of pages that works its way to the head of the Free Page List resembles the results of a LRU or NRU mechanism and the overall effect has similarities to the Second-Chance algorithm described earlier.

Another example is used by the Linux kernel on ARM. The lack of hardware functionality is made up for by providing two page tables – the processor-native page tables, with neither referenced bits nor dirty bits, and software-maintained page tables with the required bits present. The emulated bits in the software-maintained table are set by page faults. In order to get the page faults, clearing emulated bits in the second table revokes some of the access rights to the corresponding page, which is implemented by altering the native table.

Working set [edit]

Main article: Working set

The working set of a process is the set of pages expected to be used by that process during some time interval.

The "working set model" isn't a page replacement algorithm in the strict sense (it's actually a kind of medium-term scheduler)[*clarification needed*]

References [edit]

1. ^ "Lecture Notes" ⧉ by Douglas W. Jones 1995
2. ^ 2006fall:notes:lec11 [CS111] ⧉
3. ^ Characterization of Web reference behavior revisited: Evidence for Dichotomized Cache management ⧉
4. ^ 22C:116, Notes, 8 September 1995 ⧉
5. ^ Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. *Operating Systems Concepts (Seventh Edition).*: Wiley 2005. p. 339.
6. ^ VMS Help ⧉
7. ^ Andrew S. Tanenbaum. *Modern Operating Systems (Second Edition)*. pp. 218 (4.4.5). 2001.
8. ^ Sequentiality and prefetching in database systems ⧉
9. ^ "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement" by Song Jiang, Feng Chen, and Xiaodong Zhang, 2005 📄
10. ^ "WSCLOCK—a simple and effective algorithm for virtual memory management" by Richard W. Carr and John L. Hennessy, 1981 [1] ⧉ [2] ⧉
11. ^ "WSClock" ⧉ by Allan Gottlieb
12. ^ "Page Replacement Algorithms" ⧉ by Andrew S. Tanenbaum 2002
13. ^ Bansal, Sorav and Modha, Dharmendra S. (2004). "CAR: Clock with Adaptive Replacement" ⧉. *In Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. pp. 187–200. Retrieved 19 February 2012.
14. ^ Megiddo, Nimrod and Modha, Dharmendra S. (2003) *ARC: A Self-tuning, low overhead replacement cache* 📄. Proceedings of FAST '03: 2nd USENIX Conference on File and Storage Technologies
15. ^ Megiddo, Nimrod and Modha, Dharmendra S. (2004). "Outperforming LRU with an Adaptive Replacement Cache Algorithm" 📄 (PDF). *Computer* **37** (4): 58. doi:10.1109/MC.2004.1297303 ⧉.

Further reading [edit]

- K. Y. Wong, "Web Cache Replacement Policies: A Pragmatic Approach &" IEEE Network, vol. 20, iss. 2, Jan–Feb. 2006, pp. 28–34.
- Aho, Denning and Ullman, *Principles of Optimal Page Replacement* &, Journal of the ACM, Vol. 18, Issue 1,January 1971, pp 80–93
- Rhodehamel, Michael W. "The Bus Interface and Paging Units of the i860(tm) Microprocessor &". In Proc. IEEE International Conference on Computer Design, p. 380–384, 1989.
- Tanenbaum, Andrew S. *Operating Systems: Design and Implementation (Second Edition)*. New Jersey: Prentice-Hall 1997.
- Tanenbaum, Andrew S. *Modern Operating Systems (Second Edition)*. New Jersey: Prentice-Hall 2001. Online excerpt on page replacement algorithms: Page Replacement Algorithms &.
- Johnson and Shasha, *2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm* PDF (1.01 MB) abstract &
- Gideon Glass and Pei Cao Adaptive-Page-Replacement-Based-on-Memory-Reference-Behavior &. Also available in extended form as Technical Report 1338 & at www.cs.wisc.edu.

available in extended form as Technical Report 1338 at www.cs.wisc.edu

- Jongmin Kim and others, *A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References* PDF (4.14 MB), Usenix Symposium on Operating System Design and Implementation (OSDI'2000), San Diego, CA, 17–21 October 2000
- Sorav Bansal and Dharmendra S. Modha, *CAR: Clock with Adaptive Replacement* PDF (212 KB)
- Smaragdakis, Kaplan, and Wilson, *EELRU: Simple and Effective Adaptive Page Replacement* PDF (1.55 MB)
- Song Jiang and Xiaodong Zhang, *LIRS: a Low Inter Reference recency Set replacement* PDF (283 KB), SIGMETRICS 2002
- D. Lee and others, *Implementation and Performance Evaluation of the LRFU Replacement Policy*, p. 0106, 23rd Euromicro Conference: New Frontiers of Information Technology-Short Contributions, 1997
- Elizabeth J. O'Neil and others, *The LRU-K page replacement algorithm for database disk buffering* PDF (1.17 MB), ACM SIGMOD Conf., pp. 297–306, 1993.
- Y. Zhou and J.F. Philbin, *The Multi-Queue Replacement Algorithm for Second-Level Buffer Caches*, Proc. Usenix Ann. Tech. Conf. (Usenix 2001), pp. 91–104.

Categories: Virtual memory | Memory management algorithms | Online algorithms