International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2015): 78.96 | Impact Factor (2015): 6.391

Analysis of Various Page Replacement Algorithms in Operating System

Bhagyashree A. Tingare[1], **Vaishali L. Kolhe**[2]

[1, 2]D.Y. Patil College of Engineering Akurdi, Pune, Savitribai Phule Pune University, India

Abstract: ***Page replacement algorithms were a sincere topic of research and debate in the 1960s and 1970s. In a computer operating system that uses paging for virtual memory management, page replacement algorithms resolve which memory pages to page out (swap out, write to disk) when a page of memory needs to be allocated. Paging occurs when a page fault occurs and a free page cannot be used to gratify the allocation, either because there are none, or because the number of free pages i lower than some threshold. The page replacing problem is a typical online problem from the competitive analysis view in the intelligence that the optimal deterministic algorithm is known [11]. This paper is analysis on various page replacement algorithms like Optimal replacement, Random replacement, Not Recently Used (NRU), First-In, First-Out (FIFO), Least Recently Used (LRU), Second Change and CLOCK, Not Frequently Used (NFU), and some approaches like Aging, Two Queue (2Q), SEQ, Adaptive Replacement Cache (ARC), CLOCK with Adaptive Replacement (CAR), CAR with Temporal filtering (CART),Token-ordered LRU, CLOCK-Pro.***

Keywords: Optimal replacement, Random replacement, NRU, Aging, ARC, CART, Token-ordered LRU

1. Introduction

Page replacement is an important component of a modern operating system. When a page containing a desired datum or instruction is searched in translation look aside buffers or page tables and found missing from main memory, a page fault is said to occur. As the size of main memory is limited and is much smaller than the size of permanent storage, the role of page replacement is to identify the best page to evict from main memory as a result of a page fault and replace it by the a new page from disk that contains the requested datum or instruction. The problem is very similar to the block replacement in cache memories except that the page replacement is more critical as page transfers from disk to memory are orders of magnitudes slower than block transfers from main memory to the cache memory.[1] Many page replacement algorithms are used. Some of them have taken here for our comparison study. They are First-In-First-Out (FIFO), Least Recently Used (LRU), Least Recently Used with K references (LRU-K), Random, Clock with Adaptive Replacement (CAR), Adaptive Replacement Cache (ARC) and at last the most efficient rather impractical Optimal algorithm. Good replacement can reduce the page fault cost resulting in higher performance, since the more page faults the operating system encounters, the more resources are wasted on paging in/out instead of doing useful work, resulting ultimately in serious thrashing problems. In simple words we can say that-When the processor need to execute a particular page and main memory does not contain that page, this situation is known as **PAGE FAULT**. As each and every process has its own virtual address space, the operating system must keep track of all pages and the location of each page used by each process. When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache. The cache checks for the contents of the requested memory location in any cache lines that might contain that address. If the processor finds that the memory location is in the cache, a cache hit has occurred. However, if the processor does not find the memory location in the cache, a cache miss has occurred. In the case of:

1) A cache hit, the processor immediately reads or writes the data in the cache line
2) A cache miss, the cache allocates a new entry, and copies in data from main memory; then, the request is fulfilled from the contents of the cache.
   Hit ratio = Total number of Hit Counts / Total number of Reference Counts
   To represent it as a percentage:
   Hit % = Hit ratio * 100 .[11]

2. Literature Survey

1) Optimal replacement

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 [2]. 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 be 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.**

Using the formal model specified earlier, the optimal page replacement algorithm can be defined as

$$g(S, t, r_{t+1}) = \begin{cases} (S, t), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, t+1), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y has the longest time to the next reference for all pages in S. The page replacement decision depends only on the time of reference and the control state is fully described as t. In this algorithm and following algorithms, the size of S is m pages [10].

2) Random replacement

Probably the simplest page replacement algorithm is the replacement of a random page. If a frequently used page is evicted, the performance may suffer. For example, some page, that contains program initialization code which may never be needed again during the program execution, could be evicted instead. So there are performance benefits available with choosing the right page [2].

Using the formal model specified earlier, the random page replacement algorithm can be defined as

$$g(S, q_0, r_{t+1}) = \begin{cases} (S, q_0), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_0), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y is selected randomly from all pages in S. The algorithm has only one control state as the replacement decision is done identically every time [10].

3) Not Recently Used (NRU)

In the NRU algorithm [3], pages in main memory are classified based on usage during the last clock tick (See figure 1).

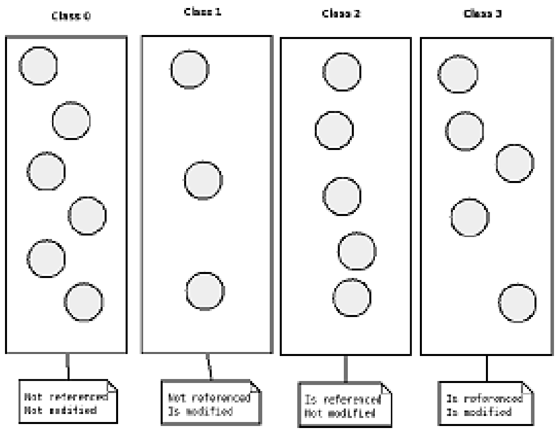

Class 0
Class 1
Class 2
Class 3
Not referenced Not modified
Not referenced Is modified
Is referenced Not modified
Is referenced Is modified

Figure 1: In NRU, the page to be evicted is selected from lowest class that contains pages

Class 0 contains pages that are not referenced nor modified, Class 1 pages that are not referenced but modified, Class 2 pages that are referenced but not modified, and Class 3 contains pages that are both referenced and modified. When a page must be evicted, NRU evicts a random page from the lowest class that contains pages.

Using the formal model specified earlier, NRU page replacement can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y is a random page from the lowest class that has pages. The control state is defined as collection of classes $q_t = \{C0_t, C1_t, C2_t, C3_t\}$. So if $C0_t = \emptyset$ and $C1_t \neq \emptyset$ page $y \in C1_t$ and $q_{t+1} = \{C0_{t+1}, C1_{t+1}, C2_t, C3_t\}$, where $C0_{t+1} = \{r_{t+1}\}$ and $C1_{t+1} = C1_t \cup r_{t+1} \{y\}$.

NRU is relatively simple to understand and implement. Implementation has a relatively low overhead, although the reference bit needs to be cleared after every clock tick. Performance is significantly better compared to pure random selection in general usage [10].

4) First-In, First-Out (FIFO)

The simple First-In, First-Out (FIFO) algorithm [3] 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 (page Z in figure 2), and the new page is inserted to head of the list (page A in figure 2). Using the formal model specified earlier, FIFO page replacement can be defined as The control state is defined as $q_t = (y_1, y_2, \dots, y_m)$.

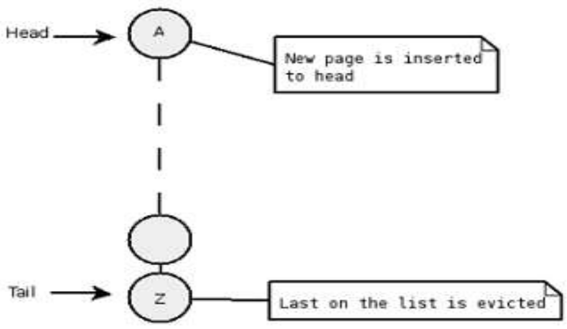


Figure 2: FIFO

Implementation of FIFO is very simple and it has a low overhead, but it is not very efficient. FIFO does not take advantage of page access patterns or frequency. Most applications do not use memory, and subsequently the pages that hold it, uniformly, causing heavily used pages to be swapped out more often than necessary [10].

5) Least Recently Used (LRU)

The Least Recently Used (LRU) [3] algorithm is based on generally noted memory usage patterns of many programs. A page that is just used will probably be used again very soon, and a page that has not been used for a long time, will probably remain unused. LRU can be implemented by keeping a sorted list of all pages in memory. The list is sorted by time when the page was last used. This list is also called LRU stack [4]. In practice this means that on every clock tick the position of the pages, used during that tick, must be updated. As a consequence, the implementation is very expensive, and not practical in its pure form. Updating on every clock tick is also an approximation, as it does not differentiate between two pages that were referenced to during the same clock tick [3].

Using the formal model specified earlier, LRU page replacement can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y_m\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

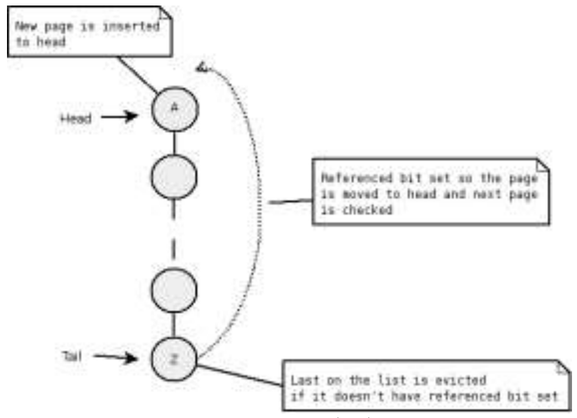
where ym is least recently used page in S. Control state is defined as

$$qt = (y1, y2, y3\dots, ym),$$

where resident pages are ordered by their most recent reference, y1 being the most recently referenced and ym the least recently referenced [10].

6) Second Change and CLOCK

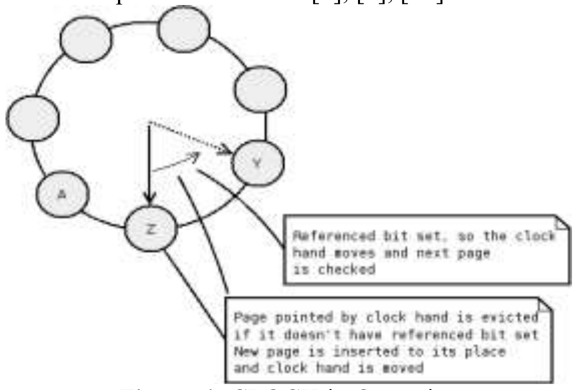
The second change [3] algorithm makes slight modification to FIFO algorithm. Instead of swapping out the last page, the referenced bit is checked. If the bit is set, the page is then moved to the head of the list as if it had just arrived and the search continues (See figure 3). If all pages are referenced then the oldest page is evicted like in FIFO.


New page is inserted to head
Head
A
Referenced bit set so the page is moved to head and next page is checked
Tail
Z
Last on the list is evicted if it doesn't have referenced bit set
Figure 3: Second Change

The CLOCK algorithm deserves a note as an implementation detail. CLOCK implements the second change so that pages are kept in a circular list with a pointer to the oldest page. When evicting, only the pointer needs to be updated and there is no need to move pages around (See figure 4). As mentioned earlier, CLOCK and its variants have been dominant in general purpose operating systems for a long time [5]. Using the formal model specified earlier, the second change algorithm can be defined as

$$g(S, q, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where yj is the first of the maintained circular FIFO list, which does not have the referenced bit set. The control state is defined as a circular FIFO list *qt = (y1, y2, ..., yi..., ym)*, where *yi* is ith most recently referenced page and pages processed before the evicted page yj are placed to the head of the list. Second change algorithm tackles the basic problem of FIFO by literally giving the page a second change before swapping it out. It can also be though as a one-bit approximation of LRU. The second change removes the problem of keeping LRU list updated, but it also shares the rest of the problems of LRU [5], [6], [10].


A
Z
Y
Referenced bit set, so the clock hand moves and next page is checked
Page pointed by clock hand is evicted if it doesn't have referenced bit set New page is inserted to its place and clock hand is moved
Figure 4: CLOCK in Operation

7) Not Frequently Used (NFU)

Not Frequently Used (NFU) [3] is another approximation of LRU. In NFU, every page has an associated usage counter which is incremented on every clock tick the page is used. When a page needs to be evicted, the page with the lowest counter value is selected. The downside of this approach is that once some process uses some pages heavily, they tend to stay there for a while, even if they are not actively used anymore. This program model of doing computation in distinct phases is very common. Also programs, that have just been started, do not get much space in the main memory as the counters start from zero.

Using the formal model specified earlier, NFU algorithm can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y_m\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y_m is the least frequently used page in S. Control state is defined as

$$q_t = ((y_1, c_1), (y_2, c_2), (y_3, c_3)\ldots, (y_m, c_m)),$$

where y_1 are resident pages ordered by the usage counters, c_i, and y_1 being the most frequently referenced page and y_1 the least frequently referenced page [10].

8) Aging

With few modifications to NFU we get an aging algorithm that is a much better approximation of LRU. Instead of incrementing an integer counter, a bit presentation of unsigned integer can be used. On every clock tick the counter value of each page is bit shifted to right, and the referenced bit of the page is inserted at the left. While the integer value of the counter is still used to make the selection of the page to be evicted, the counter value behaviour favours pages that are referenced recently, and pages that were heavily used few seconds ago, but not anymore, will get evicted sooner. The downside in this is that as the counter decrements to zero quickly, and we have no way of knowing when two pages, with zero as counter value, have been used. The other might have just been decremented to zero, while the other may have been unused for a long time. In this case a random selection, with its performance implications, is performed [3], [10].

9) Two Queue (2Q)

The two Queue, or 2Q, algorithm [7] tries to improve the detection of real hot pages and remove cold pages faster from the main memory. 2Q works by maintaining two separate lists. One is maintained as an LRU list, Hot, and the other as FIFO, F. The list F is further partitioned in to two parts Fin and Fout. The Fin list contains pages in main memory, while the Fout list contains only information of pages, not the actual contents. When page is first accessed, it is placed on the head of the Fin list. The position of the page, in the Fin list, is left untouched while it remains there. As new pages are used, Fin list will become full. When this happens the last page in Fin list is reclaimed next, but the information of the page is inserted to the head of the Fout list (page X in figure 5). If the page, now on the Fout list, is used, space is reclaimed for it and it is inserted to the head of the Hot list (page Y in figure 5). When the Fin list is not full, reclaiming is done from the tail of the Hot list. The page reclaimed from the Hot list is not inserted to any list, as it has not been used for a while (page Z in figure 5). Remember, that the Hot list is maintained as an LRU list.

The 2Q algorithm has two parameters, Kin and Kout. Kin is the maximum size of Fin and Kout is the maximum size of Fout. Authors note that setting these parameters is potentially a tuning target, but recommend reasonable values 25 % and

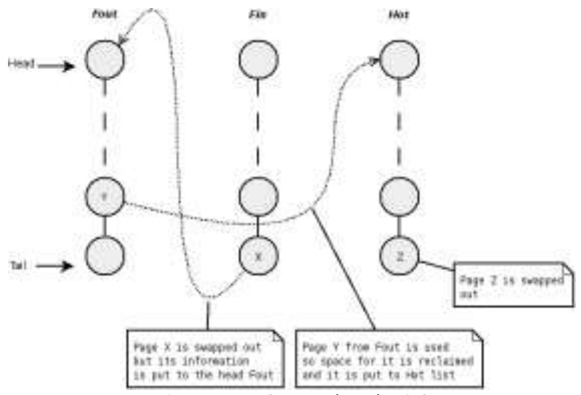

Fout
Fin
Hot
Head
Tail
Y
X
Z
Page Z is swapped out
Page X is swapped out but its information is put to the head Fout
Page Y from Fout is used so space for it is reclaimed and it is put to Hot list

Figure 5: Operation in 2Q

50 %, of page frame count, for *Kin* and *Kout*, respectively. Using the formal model specified earlier, 2Q algorithm can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y is the last page on either the Hot list or the Fin list. The control state is defined as $qt = \{Hot, Fin, Fout\}$, and each list is maintained like described earlier [10].

10) SEQ

The SEQ algorithm [8] by Gideon Glass and Pei Cao attacks rather directly against sequential memory access, an LRU unfriendly memory access pattern. SEQ works by detecting sequences of page faults within single processes address space and performs pseudo Most Recently Used (MRU) replacement. MRU tries to approximate optimal replacement algorithm. When no appropriate sequences are detected, SEQ falls back to LRU replacement. A sequence is defined by four values: PID, low, high and dir. High and low are pages with highest and lowest virtual addresses, respectively. PID identifies the process and dir identifies which direction the sequence is. When a page fault occurs, SEQ checks if the faulted page is adjacent to a sequence, with appropriate direction, belonging to the process. If so, the page is catenated to that sequence. If the extended sequence overlaps an existing sequence, the overlapped sequence is deleted. If the faulted page is in the middle of a sequence, the sequence is broken into two sequences. One sequence includes pages from the low of the sequence to page before faulted page, if direction is up, and from the page next to the faulted page, when the direction is down. The other sequence consists only of the faulted page without direction. Finally, if none of the above apply, the faulted page forms a directionless sequence by itself.

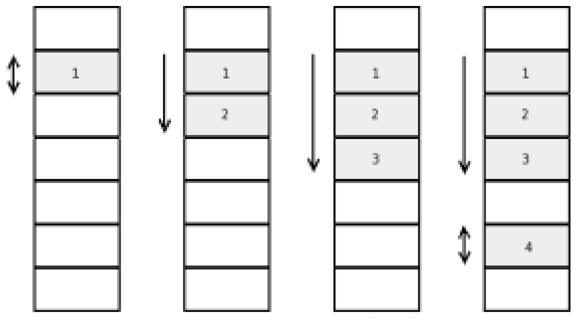


Figure 6: Sequence Detection in SEQ.

A sequence detection is shown in figure 6. The first three faulted pages are detected as a sequence, where 1 is low, 3 is high and the direction is up. The fourth page faulted is a directionless sequence. SEQ uses the detected sequences to find pages suitable for eviction. First, SEQ

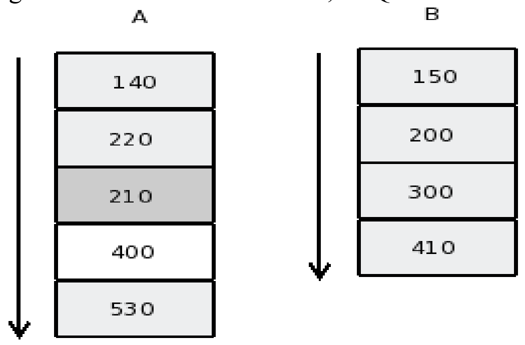


Figure 7: SEQ: Selecting a page to evict

chooses a sequence, among sequences longer than L pages, that has most recent time of the Nth most recent fault in the sequence. For example, if L is 3, N is 2 and there are two sequences presented in figure 7, sequence A is selected, because the second page of the sequence has timestamp 220 and the second page of sequence A has 200. From the selected sequence, SEQ evicts the first resident page that is at least M pages from the head of the sequence. The head of a sequence is the direction of the sequence, usually the most recent page faulted to the sequence. If M is 2, the third page from the head of sequence A is evicted (the slightly darker page in figure 7), in previous example, because the second page is already swapped out. If no suitable sequences are found, SEQ performs LRU replacement. Authors of the algorithm suggest that appropriate values for L, N and M are 20, 5 and 20, respectively [8].

Using the formal model specified earlier, SEQ algorithm can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where the state q can be defined as collection of sequences and an LRU list for fall handling, $\{lru, seq_1, seq_2, ...\}$. The page y is selected as described above [10].

11) Adaptive Replacement Cache

The Adaptive Replacement Cache (ARC) [4] algorithm, designed by Nimrod Megiddo and Dharmendra S. Modha, provides an improvement over LRU based algorithms

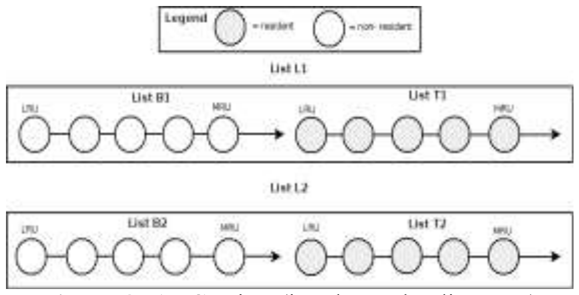


Figure 8: ARC: Lists (i.e. the cache directory)

by taking both recency and frequency into account. This is accomplished by maintaining two lists, L_1 and L_2 and remembering the history of the pages. The two lists together are called the cache directory. The list L_1 is used to capture the recency and the list L_2 to capture the frequency. Both L_1 and L_2 are kept at roughly the size of the number of page frames in the main memory *(=c)*, so the history of at most c pages, not in the main memory, is remembered. The lists L_1 *and* L_2 are partitioned into two lists, T_1, B_1, and T_2, B_2, respectively (See figure 8). T_1 contains in-cache pages that have been accessed only once, and T_2 pages that have been accessed more than once, while on lists. Consistently, list B_1 stores the history of pages evicted from the list T_1, and B_2 stores the history of pages evicted from the list T_2. The algorithm has one integer parameter, p, which is the target size for T_1. As the size of the main memory is *c*, the target size of the list T_2 is implicitly defined as c - p. This parameter p is the balance between recently used and frequently used pages. It is desirable for the algorithm to perform well under various, changing workloads and therefore ARC includes automatic adaptation of the balance between the recency and the frequency by varying the parameter p. The eviction policy in ARC is simple. If $|T_1| > p$, the least recently used page in T_1 is evicted, and if $|T_1| < p$, the least recently used page in T_2 is evicted. The eviction of a page, when $|T_1| = p$, is little more complex and, as the authors note, somewhat arbitrary. This situation is divided into three cases and it uses information of the page that caused the eviction. If the page is in B_1 or not found in B_1 *U* B_2, the least recently used page in T_2 is evicted. If the page is in *B2*, the least recently used page in T_1 is evicted. The placement policy is even simpler. If the page is found in the lists, it is placed to head of list T_2, otherwise it is placed to the head of the list T_1. When swapping in a new page (i.e. not in the lists), some history page needs to be removed. If $| T_1 \, U \, B_1 | = c$ and B_1 is not empty, the least recently used page in B_1 is removed and if B_1 is empty, the least recently used page in T_1 is swapped out and removed from the list. If $|T1 \, U \, B_1| < c$ and the history is full (i.e.$| T_1 | + | B_1 | + | T_2 | + |B_2| = 2c$), the least recently used page in B_2 is removed. The T_1 target size parameter, p, is continuously adapted to better serve the current workload. The basic idea of the adaption is to favour either recently used pages or frequently used pages. Adaptation is automatic and the direction is based on the cache hits to lists B_1 and B_2, while the amount of adaptation is based on the relative size of lists B_1 and B_2. If the requested page is found in B_1, the parameter p is increased. Likewise, if the requested page is found in B_2, the parameter p is decreased. The amount of increase is *1*.if $|B_1| \geq |B_2|$, and $|B_2|/|B_1|$ otherwise. Similarly, the amount of decrease is 1, if $|B_2| \geq |B_1|$, and $|B_1|/|B_2|$.

Using the formal model specified earlier, ARC algorithm can be defined as otherwise. Natural y, p is limited to the range [0 - c]. Increasing p means that more main memory is reserved for recently used pages, thus favouring them, while decreasing p favours frequently used pages.

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y is the least recently used page of the list T_1 or the list T_2. The state q can be defined simply as set of four lists and the target size parameter $\{T_1, B_1, T_2, B_2, p\}$ which are all maintained as described before [10].

12) CLOCK with Adaptive Replacement

CLOCK with Adaptive Replacement (CAR) [6] algorithm, and its variation CAR with Temporal filtering (CART) [6], As in ARC, cache directory of *2c* pages is kept, when the main memory can hold c pages. The directory is also partitioned to two lists L_1 and L_2, which are further partitioned to T_1 and B1, and T_2 and B_2, respectively. The lists are maintained much in the same fashion as in ARC, but there is one big difference to ARC. The strict LRU ordering of pages in T1 and T_2 is changed to a second change (or more precisely CLOCK, see figure 9). This gives the advantage of requiring only the referenced bit to be set on a page access, which is already handled by MMU, and thus action is only needed on page fault.

On page fault, the list T_1 is scanned until a page 'with the referenced bit unset is found. Let T'_1 be the pages that the scan passed. Now, the eviction policy of CAR is as follows. If $|T_1 \setminus T'_1| \geq p$, a page from T_1 is evicted. Otherwise, a page from T'_1 U T_2 is evicted. The placement policy is exactly as in ARC; if the page is not found in the history (i.e. B_1 U B_2) it is placed to T_1, otherwise it is placed to T_2. The history is managed by removing a page from B1, if $| T_1$ U B1$| = c$, and from B_2 otherwise. CAR is rather straightforward adaption of ARC for a high throughput environment, as it removes the overhead of maintaining strict LRU lists. The formal model for CAR is very similar to ARC and can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y is the least recently used page of the list T_1 or the list T_2. The state q can be defined simply as a set of four lists and the target size parameter $\{T_1, B_1, T_2, B_2, p\}$ which are all maintained as described above [10].

13) CART

CAR with Temporal filtering (CART) [6] is a variation of CAR. To CAR, CART adds a filter to better handle correlated accesses, which are typical in virtual memory. In CAR, a page is moved from T_1 to T_2 if it has been used while on T_1. CART changes this by requiring, that either $|T1| \geq min(p + 1, |B_1|)$ or the page must first enter B_1, before it can be put to T_2. This means, that a frequently used page will stay on the "recently used" list, if it is used often enough. This prevents pages with few correlated accesses to enter T_2, where it would possibly be much longer than necessary (for example, a scan that uses each page more than once). Another major difference is, that a page from T_2

is moved back to T_1, if it has the referenced bit set. These combined mean that the list T_1 acts as a temporal locality window. CART implements filtering by marking each page in the cache directory as either S, for short-term utility, or *L*, for long-term utility (See figure 10). Every page in B_1 is marked as *S* and every page in $T_2 \cup B_2$ is marked as *L*. Pages in T_1 can be marked as S or L. When a page enters the cache directory for the first time, it is marked as *S*. The page stays in T_1 as long as it has the referenced bit set when processed. While on T_1, the page is changed from *S to L if* $|T_1| \geq min(p + 1, |B_1|)$. A page marked as L is moved to T_2 if it has the referenced bit unset when processed. Pages in T_2 are moved to T_1, if they have the referenced bit set. If the page is found in $B_1 \cup B2$, it is marked as *L* and placed to T_1. Naturally, every time a page is processed, the referenced bit is unset. The history page from B_1 or B_2 is removed when $|B_1| + |B_2| = c + 1$. *If* $|B_1| >$

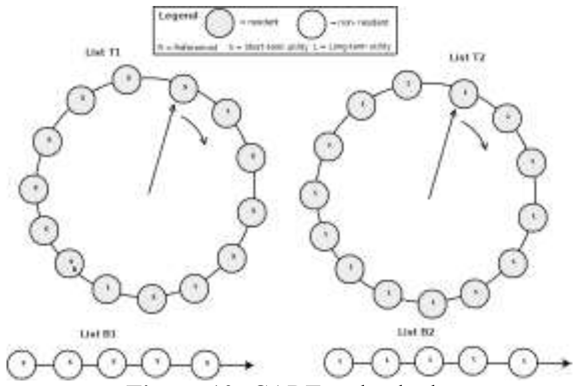


Figure 10: CART style clocks

$max(0, q)$ or B_2 is empty, the least recently used page from B_1 is removed. Otherwise, the least recently used page from B_2 is removed. The adaption in CART is done by maintaining a target size p for T_1, and additionally a target size q for B_1. Also the number of pages marked as *S* and as *L* are maintained by n_S and n_L, respectively. Like in CAR and ARC, *p* is increased when the requested page is found in B_1 and decreased when it is found in B_2. The amount of increase is $n_S/|B_1|$, *if* $n_S > |B_1|$, and *1* otherwise. Similarly, the amount of decrease is $n_L/|B_2|$, if $n_L > |B_2|$, and i otherwise. Again, the value of *p* is limited to the range *[0 - c]*. Target size *q* for the list B_1 is maintained as follows. If the requested page is found in B_2 and $|T_2| + |B_2| + |T_1| - nS \geq c$, the value of q is set as $q = min(q + 1, 2c - |T_1|)$. When moving a page from T_1 to T_2, the value of of *q* is set as $q = max(q - 1, c - |T_1|)$. The formal model for CART is very similar to ARC and can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & if\, r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_t + 1), & if\, r_{t+1} \notin S, \end{cases}$$

where *y* is the least recently used page of list T_1 or list T_2. The state *q* can be defined simply as a set of four lists and the target size parameters $\{T_1, B_1, T_2, B_2, p\}$ which are all maintained as described above [10].

14) Token-ordered LRU

Song Jiang and Xiaodong Zhang [9] noticed one significant problem in the global LRU replacement policy. A process may not be using pages belonging to its working set just because it is page faulting. A single page fault may cause one IO operation for reading a page from the secondary memory and another for writing a dirty page to it. This can lead to a significant delay in the execution of the process and to marking real working set pages of the process as candidates for eviction. Situation gets worse, if these pages are then actually evicted, as the process is effectively causing its own memory to be evicted. These working set pages, that are marked as candidates because the process is page faulting, are called false LRU pages. Eviction of these false LRU pages can cause serious trashing in the system. Token-ordered LRU [11] uses a system wide token to prevent false LRU pages from being evicted. When no main memory is available and a process tries to allocate more memory, the process grabs a token before pages for eviction are searched. Now, that candidates for eviction are searched, the pages, belonging to the process holding the token, are excluded. The pages of the token holder are strongly protected from eviction, and thus allows it to be executed with working set in main memory. This guarantees that at least one process continues to execute efficiently and prevents trashing in momentarily memory demand peaks, which are typical when, for example, system maintenance operations are performed. The token is always first taken by the process that caused the page fault. As the execution continues, the process holding the token is monitored and other processes can compete for the token. If no pages from any other process can be evicted, pages from the process holding the token are evicted and the process may lose the token. Also, if the process holds the token for too long, it is released. Overall, the target is to give the token to a short lived process or to a process that holds lots of resources in the hope that it will finish execution and release all the resources it holds. Token-ordered LRU is not actually an algorithm itself, but an addition to LRU based algorithms to prevent trashing caused by program interaction. Implementation of token-ordered LRU was officially adopted in the Linux kernel 2.6.10 [10].

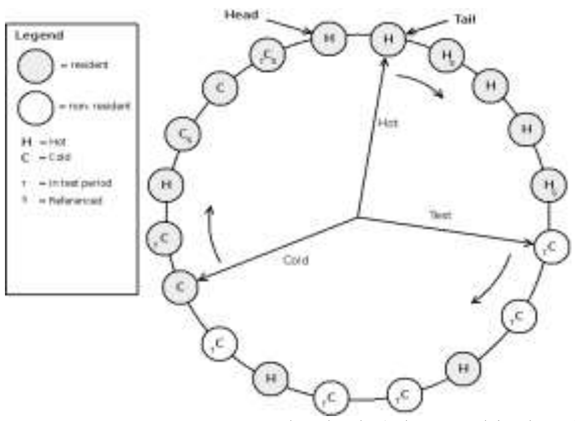


Figure 11: CLOCK-Pro style clock (The graphical style used in this thesis is adapted from [5])

15) CLOCK-Pro

CLOCK-Pro replacement algorithm [5] attacks weaknesses of LRU by changing the criteria of selecting pages for eviction, while maintaining the simple single circular list approach of CLOCK algorithm. Instead of using recency as the main criteria, as LRU does, CLOCK-Pro uses reuse distance. As discussed earlier, reuse distance is defined as the number of distinct page accesses between current access and previous access of a page. CLOCK-Pro was inspired by

LIRS [10] I/O buffer cache replacement algorithm. Using the same strategy as other algorithms, such as 2Q, ARC, CAR and CART, CLOCK-Pro keeps information of swapped out pages for some time. It uses that information to detect the reuse distance of the swapped out pages. However, instead of maintaining separate lists of resident and non-resident pages, like 2Q does, CLOCK-Pro keeps all pages in the same clock. CLOCK-Pro keeps track of all pages in the main memory and the same amount of pages that are swapped out. The resident pages are divided to two types, hot pages and cold pages. The number of hot pages is mh and the number of resident cold pages is mc. The size of the total main memory, in pages, is m, which is equal to mh + mc. Additionally, information of m non-resident pages is kept for the reuse detection. Instead of one clock hand, CLOCK Pro has three hands (figure 11): hot, cold and test.

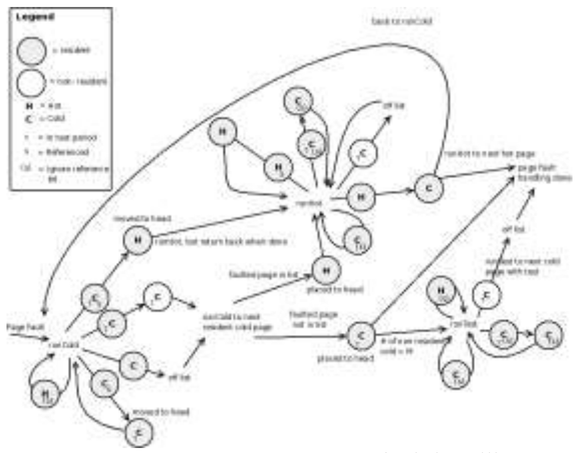


Figure 12: CLOCK-Pro page fault handling

CLOCK-Pro can be made adaptive by dynamically adjusting the balance of mc and mh. Adaption is based on current reuse distance distribution. When a cold page, whether resident or not, is accessed in its test period, the value of mc is incremented by one. When a test period of a cold page, again whether resident or not, is terminated, the value of mc is decremented by one.

Using the formal model specified earlier, CLOCK-Pro algorithm can be defined As [10]:

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

3. Conclusions

The Detail study of 15 Page replacement algorithms is been done in this paper. The growth of replacement algorithms shows the analyses and proof of better performance has moved from mathematical analysis to testing against real world program traces. This inclination shows how difficult it is to mathematically model the memory behaviour of programs. An important factor is also the large amount and easy convenience of important programs. The other clear trend is the awareness of the need for workload adaption. The simple traces used in this thesis support the inferences of the authors. Page replacement plays only a small part in overall performance of applications, but studies, have shown that the assistances are real [11].

References

[1] Development of a Virtual Memory Simulator to Analyze the Goodness of Page Replacement Algorithms Fadi N. Sibai, Maria Ma, David A. Lill
[2] L. A. Belady, A study of replacement algorithms for a virtual-storage computer, IBM Systems Journal, Volume 5, Issue 2, pp. 78–101 (1966).
[3] Andrew S. Tanenbaum and Albert S. Woodhull, Operating Systems: Design and Implementation, Third Edition, Prentice Hall, 2006.
[4] Nimrod Megiddo and Dharmendra S. Modha ARC: A Self-tuning, Low Overhead Replacement Cache USENIX File and Storage Technologies Conference (FAST), San Francisco, CA, 2003
[5] Song Jiang, Feng Chen and Xiaodong Zhang, CLOCK-Pro: An Effective Improvement of the CLOCK Replacement, USENIX Annual Technical Conference, 2005.
[6] Sorav Bansal and Dharmendra S. Modha CAR: Clock with Adaptive Replacement FAST'04 - 3rd USENIX Conference on File and Storage Technologies, 2004
[7] Theodore Johnson and Dennis Shasha. 2q: a low overhead high performance buffer management replacement algorithm In Proceedings of the Twentieth International Conference on very Large Databases, pp. 439-450, Santiago, Chile, 1994.
[8] G. Glass and P. Cao, Adaptive Page Replacement Based on Memory Reference Behavior, Proceedings of 1997 ACM SIG- METRICS Conference, May 1997, pp. 115-126.
[9] Song Jiang and Xiaodong Zhang, Token-ordered LRU: an effective page replacement policy and its implementation in Linux systems, Performance Evaluation 60 5–29, 2005.
[10] Heikki Paajanen, 'Page Replacement In Operating System Memory Management', Master's Thesis In Information Technology October 23, 2007
[11] M.Saktheeswari, K.Sridharan, 'A STUDY ON PAGE REPLACEMENT ALGORITHMS' Sri Vidya Mandir Arts & Science College Katteri, Uthangarai, International Journal of Technology and Engineering System.