APPENDIX G
VIRTUAL MEMORY PAGE REPLACEMENT ALGORITHMS

William Stallings
Copyright 2012

Supplement to
Computer Organization and Architecture, Ninth Edition
Prentice Hall 2012
ISBN: 013293633X
http://williamstallings.com/ComputerOrganization

This appendix surveys a number of page replacement algorithms for virtual memory systems. We look at four of these algorithms in some detail, and then briefly introduce a number of other algorithms. To get a feel for these algorithms, the student may want to make use of the two page replacement simulators that accompany this book.

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

We refer to the first two concepts as **resident set management**, and reserve the term **page replacement policy** for the third concept, which is discussed in this appendix.

The area of replacement policy is probably the most studied of any area of memory management. When all of the frames in main memory are occupied and it is necessary to bring in a new page to satisfy a page fault, the replacement policy determines which page currently in memory is to be replaced. All of the policies have as their objective that the page that is removed should be the page least likely to be referenced in the near future.

Because of the principle of locality, there is often a high correlation between recent referencing history and near-future referencing patterns. Thus, most policies try to predict future behavior on the basis of past behavior. One tradeoff that must be considered is that the more elaborate and sophisticated the replacement policy, the greater the hardware and software overhead to implement it.

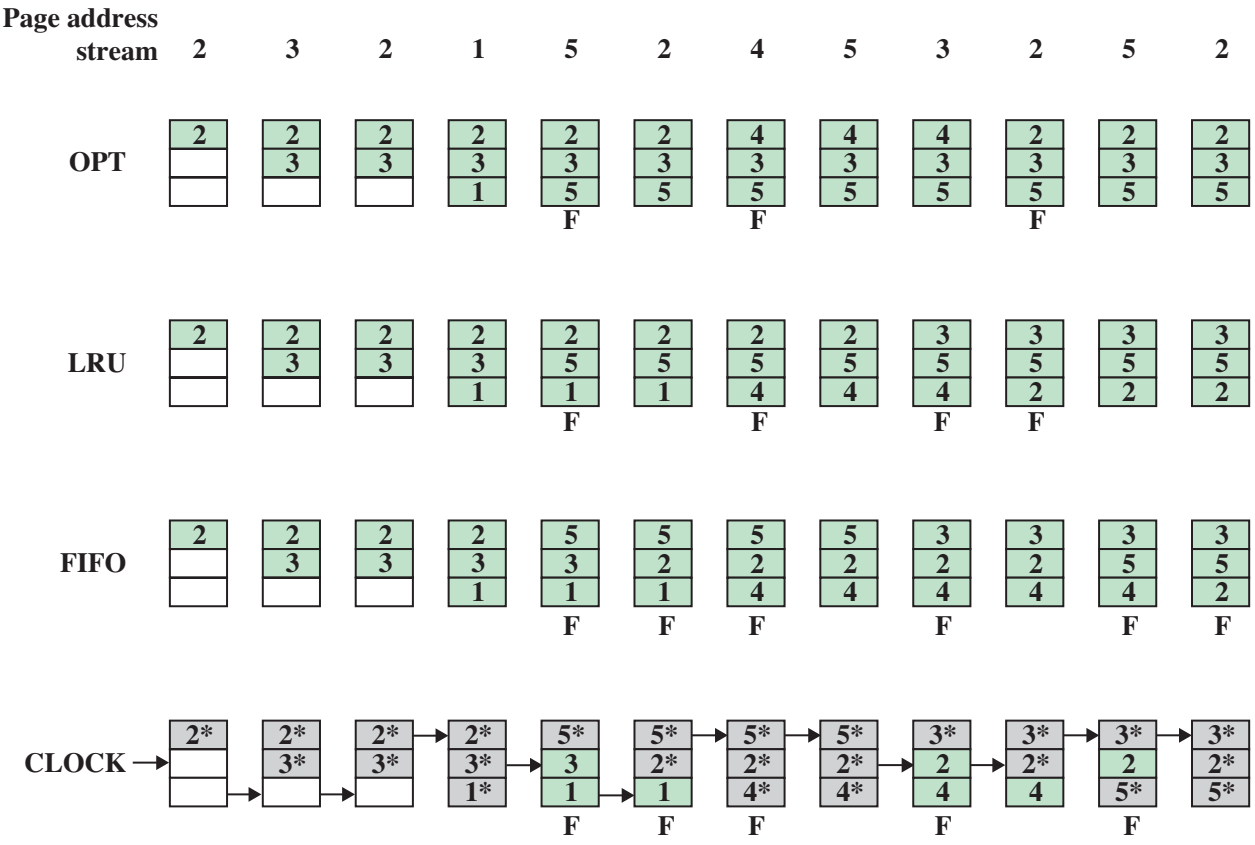
G.1 OPTIMAL

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 [BELA66]. 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.

Figure G.1 gives an example of the optimal policy. The example assumes a fixed frame allocation (fixed resident set size) for this process of three frames. The execution of the process requires reference to five distinct pages. The page address stream formed by executing the program is

2 3 2 1 5 2 4 5 3 2 5 2

which means that the first page referenced is 2, the second page referenced is 3, and so on. The optimal policy produces three page faults after the frame allocation has been filled.



F = page fault occurring after the frame allocation is initially filled

Figure G.1 Behavior of Four Page-Replacement Algorithms

G.2 LEAST RECENTLY USED

The **least recently used** (LRU) policy replaces the page in memory that has not been referenced for the longest time. By the principle of locality, this should be the page least likely to be referenced in the near future. And, in fact, the LRU policy does nearly as well as the optimal 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. Even if the hardware would support such a scheme, the overhead would be tremendous.

Alternatively, one could maintain a stack of page references, again an expensive prospect.

Figure G.1 shows an example of the behavior of LRU, using the same page address stream as for the optimal policy example. In this example, there are four page faults.

G.3 FIRST-IN-FIRST-OUT

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.

Continuing our example in Figure G.1, the FIFO policy results in six page faults. Note that LRU recognizes that pages 2 and 5 are referenced more frequently than other pages, whereas FIFO does not.

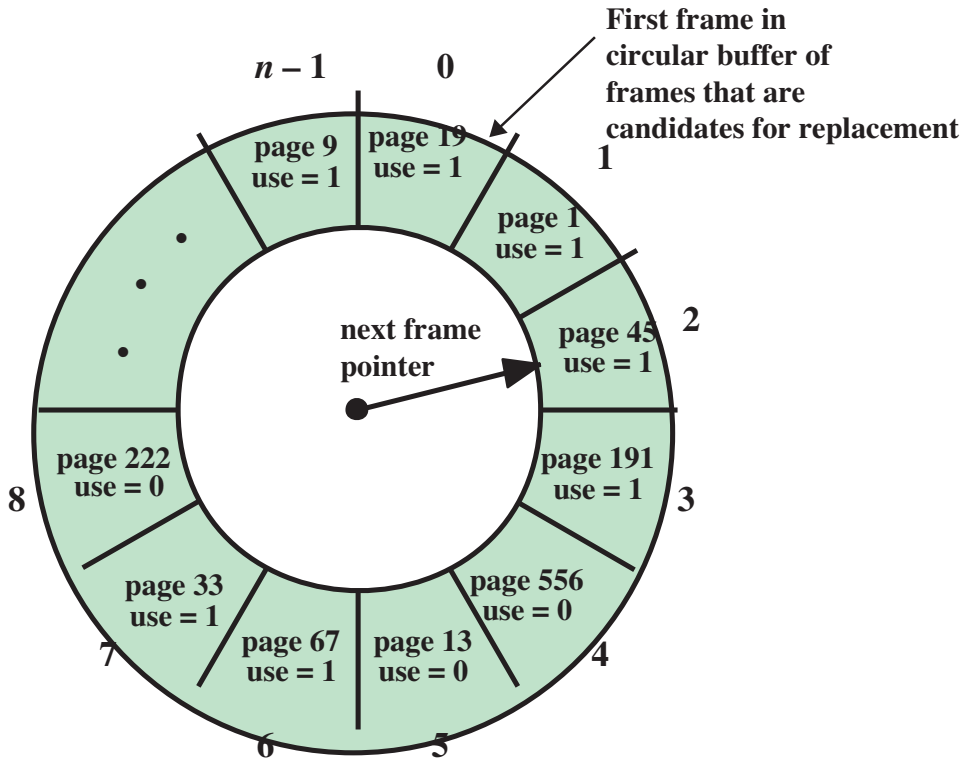
Although the LRU policy does nearly as well as an optimal policy, it is difficult to implement and imposes significant overhead. On the other hand, the FIFO policy is very simple to implement but performs relatively poorly. Over the years, operating system designers have tried a number of other algorithms to approximate the performance of LRU while imposing little overhead. Many of these algorithms are variants of a scheme referred to as the **clock policy**.

The simplest form of clock policy requires the association of an additional bit with each frame, referred to as the use bit. When a page is first loaded into a frame in memory, the use bit for that frame is set to 1. Whenever the page is subsequently referenced (after the reference that generated the page fault), its use bit is set to 1. For the page replacement algorithm, the set of frames that are candidates for replacement (this process: local scope; all of main memory: global scope[1]) is considered to be a circular buffer, with which a pointer is associated. When a page is replaced, the pointer is set to indicate the next frame in the buffer after the one just updated. When it comes time to replace a page, the operating system scans the buffer to find a frame with a use bit set to zero. Each time it encounters a frame with a use bit of 1, it resets that bit to zero and continues on. If any of the frames in the buffer have a use bit of zero at the beginning of this process, the first such frame encountered is chosen for replacement. If all of the frames have a use bit of 1, then the pointer will make one complete cycle through the buffer, setting all the use bits to zero, and stop at its original position, replacing the page in that frame. We can see that this policy is similar to FIFO, except that, in the clock policy, any frame with a use bit of 1 is passed over by the algorithm. The policy is referred to as a clock policy because we can visualize the page frames as laid out in a circle. A number of operating systems have employed some variation of this simple clock policy (for example, Multics [CORB68]).

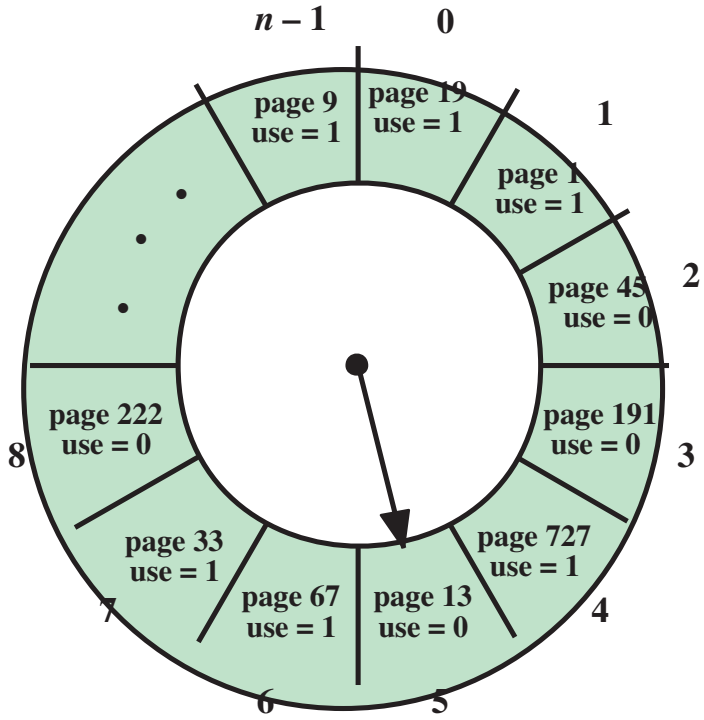
Figure G.2 provides an example of the simple clock policy mechanism. A circular buffer of *n* main memory frames is available for page replacement. Just prior to the replacement of a page from the buffer with incoming page 727, the next frame pointer points at frame 2, which contains page 45. The clock policy is now executed. Because the use bit for page 45 in frame 2 is

---

[1] The concept of scope is discussed in the subsection "Replacement Scope," subsequently.



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

Figure G.2 Example of Clock Policy Operation

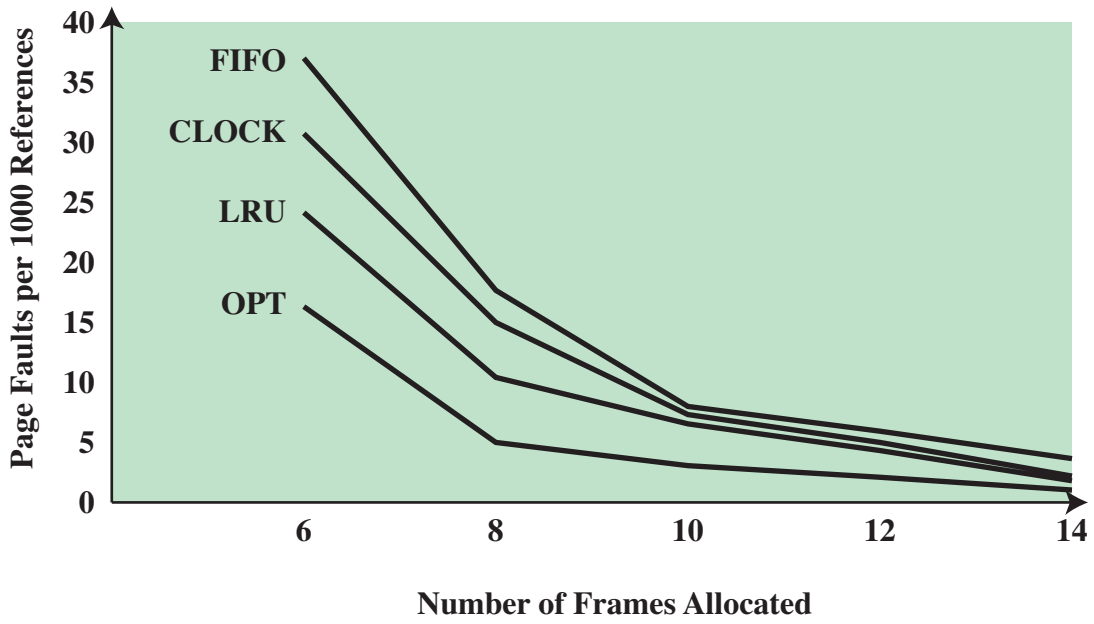


Figure G.3 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

equal to 1, this page is not replaced. Instead, the use bit is set to zero and the pointer advances. Similarly, page 191 in frame 3 is not replaced; its use bit is set to zero and the pointer advances. In the next frame, frame 4, the use bit is set to 0. Therefore, page 556 is replaced with page 727. The use bit is set to 1 for this frame and the pointer advances to frame 5, completing the page replacement procedure.

The behavior of the clock policy is illustrated in Figure G.3. The presence of an asterisk indicates that the corresponding use bit is equal to 1, and the arrow indicates the current position of the pointer. Note that the clock policy is adept at protecting frames 2 and 5 from replacement.

Figure G.3 shows the results of an experiment reported in [BAER80], which compares the four algorithms that we have been discussing; it is assumed that the number of page frames assigned to a process is fixed. The results are based on the execution of 0.25×10^6 references in a FORTRAN program, using a page size of 256 words. Baer ran the experiment with

frame allocations of 6, 8, 10, 12, and 14 frames. The differences among the four policies are most striking at small allocations, with FIFO being over a factor of 2 worse than optimal. In order to run efficiently, we would like to be to the right of the knee of the curve (with a small page fault rate) while at the same time keeping a small frame allocation (to the left of the knee of the curve). These two constraints indicate that a desirable mode of operation would be at the knee of the curve.

Almost identical results have been reported in [FINK88], again showing a maximum spread of about a factor of 2. Finkel's approach was to simulate the effects of various policies on a synthesized page-reference string of 10,000 references selected from a virtual space of 100 pages. To approximate the effects of the principle of locality, an exponential distribution for the probability of referencing a particular page was imposed. Finkel observes that some might be led to conclude that there is little point in elaborate page replacement algorithms when only a factor of 2 is at stake. But he notes that this difference will have a noticeable effect either on main memory requirements (to avoid degrading operating system performance) or operating system performance (to avoid enlarging main memory).

The clock algorithm has also been compared to these other algorithms when a variable allocation and either global or local replacement scope (see the following discussion of replacement policy) is used [CARR81, CARR84]. The clock algorithm was found to approximate closely the performance of LRU.

G.4 OTHER PAGE REPLACEMENT ALGORITHMS

We now briefly introduce some alternative page replacement algorithms.

Random

This algorithm picks a page at random to replace. We can think of this as a minimal requirement. Any algorithm that fails to improve on random replacement is clearly undesirable.

Least Frequently Used (LFU)

This algorithm replaces the page that has been used least frequently. The rationale for this approach is similar to that for the LRU algorithm. The overhead is that of keeping a use count for each page. One drawback is that a recently loaded page will in general have a low use count and may be replaced inadvisably. A way to avoid this is to inhibit the replacement of pages loaded with the last given time interval.

Least Recently Used - K (LRU-K)

LRU-K combines some of the features of both LRU and LFU. A reference history is kept that captures the last *K* references for each page. The page to be replaced is the page whose the *backward K distance* of a page is the distance (number of page references overall) back to the K^{th} most recent reference of the page. The page with the largest backward *K* distance is the one chosen for replacement. That is, the algorithm determines the K^{th} most recent reference to each page in main memory. The page with the oldest such reference is chosen for replacement. Note that LRU-1 reduces to LRU. The LRU-K algorithm, unlike LRU, takes into account frequency of use. However, LRU-K is also different from LFU. The key difference is that LRU-K has a built-in notion of *aging*, considering only the last K references to a page, whereas the LFU algorithm has no means to discriminate recent versus past reference frequency of a page.

Adaptive Replacement Cache (ARC)

ARC is another attempt to combine the recency concept of LRU and the frequency concept of LFU. This is a relatively complex algorithm. For this appendix, we quote the summary given in [MEGI03]. For more detail, see [MEGI04].

Suppose that a cache can hold c pages. The ARC scheduler maintains a cache directory that contains 2c pages, c pages in the cache and c history pages. ARC's cache directory, referred to as DBL, maintains two lists: L1 and L2. The first list contains pages that have been seen only once recently, while L2 contains pages that have been seen at least twice recently. The replacement policy for managing DBL is: Replace the LRU page in L1 if it contains exactly c pages; otherwise, replace the LRU page in L2.

The ARC policy builds on DBL by carefully selecting c pages from the 2c pages in DBL. The basic idea is to divide L1 into a top T1 and bottom B1 and to divide L2 into top T2 and bottom B2. The pages in T1 are more recent than those in B1; likewise for T2 and B2. The algorithm includes a target size target_T1 for the T1 list. The replacement policy is simple: Replace the LRU page in T1, if T1 contains at least target_T1 pages; otherwise, replace the LRU page in T2.

The adaptation comes from the fact that the target size target_T1 is continuously varied in response to an observed workload. The adaptation rule is also simple: Increase target_T1, if a hit in the history B1 is observed; similarly, decrease target_T1, if a hit in the history B2 is observed.

Clock with Adaptive Replacement (CAR)

CAR is a refinement of the Clock algorithm using the principles of ARC [BANS04]. 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 are first inserted in

T1 and graduate to T2 upon passing a certain test of long-term utility. By using a certain precise history mechanism that remembers recently evicted pages from T1 and T2, the algorithm adaptively determine the sizes of these lists in a data-driven fashion.

References

BANS04 Bansal, S., and Dharmendra, S. "CAR: Clock with Adaptive Replacement." Proceedings, *USENIX File and Storage Technologies (FAST)*, 2004.

MEGI03 Megiddo, N., and Dharmendra, S. "One up on LRU." *;login*, August 2003.

MEGI04 Megiddo, N., and Dharmendra, S. "Outperforming LRU with an Adaptive Replacement Cache Algorithm." *IEEE Computer Magazine*, April 2004.