Virtual Memory Page Replacement Algorithms

Terrance McCraw
CS384 – Operating Systems
Milwaukee School of Engineering
mccrawt@msoe.edu

Abstract

Virtual memory has become a necessity over the years as the cost of secondary memory decreases, and the memory demands of software increase. To balance the use of primary memory among processes, sophisticated algorithms are necessary to manage the replacement of virtual pages. This paper outlines the pros and cons of a variety of solutions, including both static demand based and dynamic prefetch replacement algorithms. Ultimately, the effectiveness of any algorithm is dependent upon the application and hardware support available, although some generalization about performance can be made.

Virtual Memory Introduction

Virtual memory is a catch-all phrase for the abstraction of physical memory via a virtual address space. In all cases a virtual memory manager is responsible for maintaining this virtual address translation; generally speaking the definition also includes the responsibility of operating a large virtual address range within a smaller available physical memory. This process involves the transfer of blocks of memory from a secondary source (usually a slower hard-disk) to the primary memory whenever necessary for program execution. It is the virtual memory manager's responsibility to provide a level of abstraction for programs, allowing them to operate seamlessly without any “knowledge” of the underlying system. If implemented properly, software can be written for these systems using a relatively large virtual address range, while running on a small amount of physical memory with little reduction in speed.

Segmentation and Paging

At its very roots virtual addressing is applied one of two ways: either via segmentation or paging. Segmentation involves the relocation of variable sized segments into the physical address space. Generally these segments are contiguous units, and are referred to in programs by their segment number and an offset to the requested data. Although a segmentation approach can be more powerful to a programmer in terms of control over the memory, it can also become a burden, as suggested by [1]. Efficient segmentation relies on programs that are very thoughtfully written for their target system. Even assuming best case scenarios, segmentation can lead to problems, however.

As described by [2], external fragmentation is the term coined for pieces of memory between segments, which may collectively provide a useful amount of memory, but are rendered useless by their non-contiguous nature. Since segmentation relies on memory that is located in single large blocks, it is very possible that enough free space is available to load a new module, but cannot be utilized. Segmentation may also suffer from internal fragmentation if segments are not variable-sized, where memory above the segment is not used by the program but is still “reserved” for it.

Contrarily, paging provides a somewhat easier interface for programs, in that its operation tends to be more automatic and thus transparent. Each unit of transfer, referred to as a page, is of a fixed size and swapped by the virtual memory manager outside of the program’s control. Instead of utilizing a segment/offset addressing approach, as seen in segmentation, paging uses a linear sequence of virtual addresses which are mapped to physical memory as necessary, evidenced in [1,3]. Due to this addressing approach, a single program may refer to a series of many non-contiguous segments. Although some internal fragmentation may still exist due to the fixed size of the pages, the approach virtually eliminates external fragmentation. According to [3], the advantages of paging over segmentation generally outweigh their disadvantages.

Implementation Feasibility

Virtual memory is good in theory, but for its operation to be practical it must be properly implemented. Algorithms responsible for replacing pages in physical memory from the secondary source are primarily responsible for the speed and efficiency of the final system. To operate effectively, the loading of extraneous information must be minimized or completely eliminated, lest the manager’s use of the resources become wasteful. Further information on efficient swap methods can be found in [4]. More importantly, information that is swapped out of the physical memory must be chosen carefully. If a page has been removed from memory to make way for another requested page, but then is immediately requested once again, we say the replacement is thrashing. Thrashing page replacement has the potential to bring virtual memory to an immediate slowdown suggests [5], since it causes the manager to make redundant memory reads and writes, while relying heavily on the speed of the secondary storage device.

Thankfully most thrashing can be avoided naturally, as a program’s scope of operation tends to remain relatively small throughout its lifetime. This idea, the principal of locality, states that program code and data references will most likely not be contiguous, but will reliably cluster in predictable areas. Without the clustering behavior of pages, “predictive non-thrashing algorithms could not function”, states [3]. Aware of these principals, we can begin evaluating the variety of page replacement algorithms.

Demand/Prefetch Fetching Policies

Upon initial operation [1] suggests we can assume that the paging mechanism will have no prior knowledge of the page reference stream, or the order pages will be requested in. This causes many systems to employ a demand fetch approach, where a page fault notification is the first indication that a page must be moved into the physical memory. Prefetch, or dynamic page replacement is also possible, and will be examined after static algorithms.

All paging algorithms function on three basic policies: a fetch policy, a replacement policy, and a placement policy. In the case of static paging, [1] describes the process with a shortcut: the page that has been removed is always replaced by the incoming page; this means that the placement policy is always fixed. Since we are also assuming demand paging, the fetch policy is also a constant; the page fetched is that which has been requested by a page fault. This leaves only the examination of replacement methods.

Static Page Replacement Algorithms

Optimal Replacement Theory

In a best case scenario the only pages replaced are those that will either never be needed again, or have the longest number of page requests before they are referenced. This "perfect" scenario is usually used only as a benchmark by which other algorithms can be judged, and is referred to as either Belady's Optimal Algorithm, as described by [1,5] or Perfect Prediction (PP), as seen in [7]. Such a feat cannot be accomplished without full prior knowledge of the reference stream, or a record of past behavior that is incredibly consistent. Although usually a pipe dream for system designers, [1] suggests it can be seen in very rare cases, such as large weather prediction programs that carry out the same operations on consistently sized data.

Random Replacement

On the flip-side of complete optimization is the most basic approach to page replacement: simply choosing the victim, or page to be removed, at random. Each page frame involved has an equal chance of being chosen, without taking into consideration the reference stream or locality principals. Due to its random nature, the behavior of this algorithm is quite obviously, random and unreliable. With most reference streams this method produces an unacceptable number of page faults, as well as victim pages being thrashed unnecessarily. As commented on by [7], better performance can almost always be achieved by employing a different algorithm. Most systems stopped experimenting with this method as early as the 1960's [1].

First-In, First-Out (FIFO)

First-in, first-out is as easy to implement as Random Replacement, and although its performance is equally unreliable or worse, claims [7], its behavior does follow a predictable pattern. Rather than choosing a victim page at random, the oldest page (or first-in) is the first to be removed. Conceptually [4] compares FIFO to a limited size queue, with items being added to the queue at the tail. When the queue fills (all of the physical memory has been allocated), the first page to enter is pushed out of head of the queue. Similar to Random Replacement, FIFO blatantly ignores trends, and although it produces less page faults, still does not take advantage of locality trends unless by coincidence as pages move along the queue [1].

A modification to FIFO that makes its operation much more useful is First-In Not-Used First-Out (FINUFO). The only modification here is that a single bit is used to identify whether or not a page has been referenced during its time in the FIFO queue. This utility, or referenced bit, is then used to determine if a page is identified as a victim. If, since it has been fetched, the page has been referenced at least once, its bit becomes set. When a page must be swapped out, the first to enter the queue whose bit has not been set is removed; if every active page has been referenced, a likely occurrence taking locality into consideration, all of the bits are reset. In a worst-case scenario this could cause minor and temporary thrashing, but is generally very effective given its low cost [7]. Further information on reference bits and their application to other algorithms can be found in [2].

Least Recently Used (LRU)

We have seen that an algorithm must use some kind of behavior prediction if it is to be efficient [3]. One of the most basic page replacement approaches uses the usage of a page as an indication of its "worth" when searching for a victim page: the Least Recently Used (LRU) Algorithm. LRU was designed to take advantage of "normal" program operation, which generally consists of a series of loops with calls to rarely executed code [1]. In terms of the virtual addressing and pages, this means that the majority of code executed will be held in a small number of pages; essentially the algorithm takes advantage of the locality principal.

As per the previous description of locality, LRU assumes that a page recently referenced will most likely be referenced again soon. To measure the "time" elapsed since a page has been a part of the reference stream, a backward distance is stored [2]. This distance must always be greater than zero, the point for the current position in the reference stream, and can be defined as infinite in the case of a page that has never been referenced. Thus, the victim page is defined as the one with the maximum backward distance; if two or more points meet this condition, a page is chosen arbitrarily. This process is described in detail with numerical examples in [1].

Actual implementation of the backward distance number can vary, and does play an important role in the speed and efficiency of this algorithm. This can be done by sorting page references in order of their age into a stack, allowing quick identification of victims [2]. However the overhead associated with sorting does not generally justify the speed of identification, unless specific hardware exists to perform this operation. Many operating systems do not assume this hardware exists (such as UNIX), and instead increment an age counter for every active page during the page stream progression, as described by [7]. When a page is referenced once again, or is brought in due to a page fault, its value is simply set to zero. Since storage for the backward age is limited, a maximum value may also be defined; generally any page that has reached this age becomes a valid target for replacement [4]. As with any algorithm, modifications can be made to increase performance when additional hardware resources are available.

Additional information about more complex LRU algorithms can found in [11].

<u>Least Frequently Used (LFU)</u>

Often confused with LRU, Least Frequently Used (LFU) selects a page for replacement if it has not been used often in the past. Instead of using a single age as in the case of LRU, LFU defines a frequency of use associated with each page. This frequency is calculated throughout the reference stream, and its value can be calculated in a variety of ways.

The most common frequency implementation begins at the beginning of the page reference stream, and continues to calculate the frequency over an ever-increasing interval. Although this is the most accurate representation of the actual frequency of use, it does have some serious drawbacks. Primarily, reactions to locality changes will be extremely slow [1]. Assuming that a program either changes its set of active pages, or terminates and is replaced by a completely different program, the frequency count will cause pages in the new locality to be immediately replaced since their frequency is much less than the pages associated with the previous program. Since the context has changed, and the pages swapped out will most likely be needed again soon (due to the new program's principal of locality), a period of thrashing will likely occur.

If the beginning of the reference stream is used, initialization code of a program can also have a profound influence, as described by [1]. The pages associated with initial code can influence the page replacement policy long after the main body of the program has begun execution. One way to remedy this is to use a popular variant of LFU, which uses frequency counts of a page since it was last loaded rather than since the beginning of the page reference stream. Each time a page is loaded, its frequency counter is reset rather than being allowed to increase indefinitely throughout the execution of the program. Although this policy will for the most part prevent “old” pages from having a huge influence in the future of the stream, it will still tend to respond slowly to locality changes [1].

Stack Algorithms

One would naturally expect the behavior of static paging algorithms to be linear; after all, they are static in nature. Instinct tells us that by increasing the available physical memory for storing pages, and thus decreasing the needed number of page replacements, that the performance of the algorithm would increase. However, with most simple algorithms this is not necessarily the case. In fact, by increasing the available physical memory, some algorithms such as FIFO can decrease in page fault performance seemingly at random, as evidenced by [1]. This occurrence is referred to as Belady’s Anomaly, and is a primary factor in considering the practicality of any static algorithm [1,2,5].

The predictable change in performance with an increase in physical memory is obviously not something to be taken for granted. It can be proven, however, that if any algorithm with allocation of size *m* has pages that are guaranteed to be a subset of the allocation *m* + 1, it will not be subject to Belady’s Anomaly; this is what is referred to as the inclusion property [1,2]. Static algorithms that meet this requirement are called Stack Algorithms, named rightly so for the process of stacking subsets of pages as available allocations increase.

Not only are Stack Algorithms more useful, since they are guaranteed not to degrade in performance as available resources increase, their page faulting behavior is also easy to predict:

> “For example, one can calculate the cost of page fetches with a single pass over the reference stream for a stack algorithm, since it is possible to predict the number of page faults by analyzing the memory state. Also, the memory state can be used to predict performance improvement obtained by increasing a process’s memory allocation for stack algorithms. This performance improvement is not possible for other algorithms.” [1]

Examples of Stack Algorithms include LRU and LFU, which are among the minority of algorithms not subject to Belady’s Anomaly.

Dynamic Page Replacement Algorithms

All of the static page replacement algorithms considered have one thing in common: they assumed that each program is allocated a fixed amount of memory when it begins execution, and does not request further memory during its lifetime. Although static algorithms will work in this scenario, they are hardly optimized to handle the common occurrence of adjusting to page allocation changes. This can lead to problems when a program rapidly switches between needing relatively large and relatively small page sets or localities [1].

Depending on the size of the memory requirements of a program, the number of page faults may increase or decrease rapidly; for Stack Algorithms, we know that as the memory size is decreased, the number of page faults will increase. Other static algorithms may become completely unpredictable. Generally speaking, any program can have its number of page faults statistically analyzed for a variety of memory allocations. At some point the rate of increase of the page faults (derivative of the curve) will peak; this point is sometimes referred to as the hysteresis point [1]. If the memory allocated to the program is less than the hysteresis point, the program is likely to thrash its page replacement. Past the point, there is generally little noticeable change in the fault rate, making the hysteresis the target page allocation [1,6].

Since a full analysis is rarely available to a virtual memory controller, and that program behavior is quite dynamic, finding the optimal page allocation can be incredibly difficult. A variety of methods must be employed to develop replacement algorithms that work hand-in-hand with the locality changes present in complex programs. Dynamic paging algorithms accomplish this by attempting to predict program memory requirements, while adjusting available pages based on reoccurring trends. This policy of controlling available pages is also referred to as “prefetch” paging, and is contrary to the idea of demand paging [5].

Although localities (within the scope of a set of operations) may change, states [4], it is likely that within the global locality (encompassing the smaller clusters), locality sets will be repeated. This idea of a “working set” of localities is mentioned in [1-6,8], and is the basis for most modern operating systems’ replacement algorithms [8].

Working Set Algorithms

Working Set Replacement (WSR)

Mathematically speaking Working Set Replacement (WSR) algorithms can either be very simple or extremely complex. Essentially, the most basic algorithms assume that each program will use only a limited number of its pages during a certain interval of time. During this interval, the program is allowed to freely page fault and add pages, growing until the time has expired [1]. When the interval has expired, the virtual memory manager removes all page references unused during the previous interval. We refer to the set of pages used by a program during its previous interval as its working set [2,6]. For this to work reliably with minimal thrashing, the time elapsed may be dynamically adjusted to provide maximal correspondence with locality changes. These adjustments can be made a variety of ways, but are usually determined as a function of the rate of page faults occurring within the program, as touched upon in [1].

Page-Fault Frequency (PFF)

Working set algorithms do not always use a specific time interval to determine the active set. Various page fault frequency (PFF) algorithms can also be used to monitor the rate at which a program incurs faults [7]. This is very similar to modifying the time interval but is not subject to a minimal time for change to occur; page allocation or release may occur rapidly during periods of locality transition, rather than attempting to suddenly minimize the time interval for evaluation to accomplish the same goal. It is these types of dynamic changes that can add complexity to the working set implementation.

PFF does have its limitations depending on the application, however. An example program, given by [7], may require unrelated references to a database, causing a large fault frequency. In this scenario, the program would not benefit from keeping the old references in memory. Rapid changes in the fault frequency due to this type of access would result in either wasted page allocation or rapid thrashing with this algorithm, both detracting from its usefulness. More often than not these types of unrelated memory references are an uncommon occurrence, however.

Clocked LRU Approximation / WSClock

There are other working set methods that closely approximate static methods, only on a global scale. One such algorithm is the Clocked LRU Approximation. Clock algorithms generally operate by envisioning the page frames arranged circularly, such as on a clock face [6]. Frames are considered for replacement by a pointer moving clockwise along the virtual clock face; at this point that particular frame is evaluated on some criteria, and the page is either replaced or the pointer moves on.

To simulate LRU with the clocked method, the page table entry's valid bit is used by the system as a software settable reference bit, as described previously. It also relies on the ability to have the hardware check a software based valid bit instead of its default in the page table. A system clock routine periodically runs through each program's page table, resetting the valid bits. If a page is referenced, a page fault will occur, and the hardware will reference the software valid bit, discovering that the page is in memory. If it is, the page fault hander will set the page table entry's valid bit and continue the process. When the clock process is allowed to run again, if it finds an invalid page with its software bit set and its page table entry's valid bit not set, the operating system knows that the page has not been referenced; it can be assumed that this page is no longer a part of the working set and can be removed [6,7]. The set of algorithms that the Clocked LRU belongs to are called WSClock, meaning working set clock. Although WSClock generally behaves similar to LRU, its actual performance can differ based on timing parameters and selection criteria [1].

Replacement Algorithm Evaluation

Methodology of Evaluation

It quickly becomes obvious when evaluating algorithms that a common benchmark is difficult to identify; although standards such as Belady's

Optimal Algorithm or Perfect Prediction can be used as performance benchmarks across static and dynamic algorithms, they are certainly not the end-all of judgment. One of the largest concerns when comparing algorithms is not only their speed, but the relative cost in resources necessary to accomplish them. Depending on the implementation, many algorithms may not even be feasible due to hardware restrictions, or be subject to performance decreases due to limited hardware support [4]. Other design considerations, such as the size of pages present on the system can also affect algorithm performance. Statistical evidence of the huge effect on performance due to page size can be found in [9].

For any comparison to be feasible a few generalities must be made. We can assume that a sufficient architecture for implementing virtual memory does exist, such as a virtual memory manager, and that a single page replacement generally takes some fixed average time. In reality, page reads from a secondary memory may vary widely due to the storage medium or system-wide demand for access to that device, alluded to by the relatively slow performance of secondary devices in [10].

Static Algorithm Comparisons

Despite any necessary hardware requirements Static Algorithms are subject to one critical criterion described previously: they are subject to inclusion and are thus a Stack Algorithm. Methods that do not belong to the Stack distinction, identified by [1] as Random Replacement and FIFO, can be put to little use unless their reactions are known for a specific subset of hardware that will not provide additional resources beyond the test conditions. Unfortunately randomized conditions result in predictable outcomes only if it truly does not matter which pages are removed, which is almost never the case.

Although not the most effective algorithm in applications with a variety of operations, FIFO may perform well with relatively little cost if operation is very consistent. Note that this can only be established by experimental findings; FIFO performing well is dependent on consistent operation. Consistent operation is not necessarily indicative of FIFO being advantageous, however. In general any situation that employs FIFO successfully would still be better with a modified version, such as FINUFO. FINUFO provides a low cost solution much like FIFO, but performs almost as well as more complex algorithms, such as LRU [7]. This approach does require minimal additional hardware support.

In terms of approaching Belady's Optimal Algorithm, LRU performance is one of the most effective in the Static Algorithm subset.

> "… LRU has become the most widely used of the static replacement algorithms because it is a reasonable predictor for program behavior and produces good performance on a wide variety of page reference streams." [1]

In addition to the LRU Algorithm's excellent performance, it also falls into the Stack Algorithm category, making its performance quite predictable and suitable for use on scalable systems. However for LRU to work correctly, it requires additional hardware support in the form of a time field for each active page. Without a portion of hardware devoted to updating the time, an interrupt would have to be used to run a manual routine. This approach, as described by [2], suffers from a memory reference time increase by a factor of ten or more, "hence slowing every user process by a factor of ten".

LFU presents the same general advantages as LRU, such as being a Stack Algorithm, and additionally provides a more accurate means of determining which pages in memory are useful. LFU does suffer from pitfalls, however, and has a difficult time adjusting to locality changes as previously mentioned, which may be common in large systems [1]. Although it would seem that this would make LFU ideal for more specialized applications, it comes at the cost of needing even more complex hardware than LRU, to perform the frequency calculations necessary. Although both are excellent algorithms, they are rarely seen without some approximations due to their hardware demands [2,7].

Dynamic Algorithm Comparisons

Algorithms based upon prefetching pages can be judged with some of the same standards as demand based Static Algorithms. Performance wise algorithms based on the working set principal may very greatly. Aside from the overhead involved with switching localities on a global scale, working set approaches can use a variety of methods to choose their victim pages, much the same as Static Algorithms. Thus, working set approaches can generally be gauged not by their criterion for local page replacement, but by their conditions for working page set replacement.

Selection of intervals for WSR can be a costly process, but usually requires only a minimal amount of timing hardware and the ability to monitor page faults. More elaborate WSR solutions using PFF become increasingly costly as their ability to accurately measure page fault rate changes, as described in [6]. As with Static Algorithms, approximations such as Clocked LRU can provide nearly the same level of performance with significantly less hardware cost. Generally this type of an approach is preferred over the direct implementation.

Conclusions

An investigation of virtual memory proves that its concept is not only feasible, but extremely useful and a necessity in ever-growing computer systems where high speed primary memory is limited. Although not the only factor to be considered in the effectiveness of a virtual memory controller, replacement algorithms play one of the most vital roles in the overall performance of such a system, attempting to minimize redundant access to slower secondary storage. Since a program's page reference stream is almost never known, algorithms that take into consideration past trends are a necessity for good performance.

Although it is apparent that Dynamic Algorithms are more versatile in their ability to deal with locality changes and the natural occurrence of working page set changes, their complexity makes them a reality only for large-scale systems. When working with smaller systems, approximations of Static Algorithms such as Least Recently Used (LRU) or Least Frequently Used (LFU) tend to yield the best performance while dealing with limited hardware support for additional functions; direct application of these algorithms generally requires too much hardware overhead to be practical. Aside from performance, it is the predicable nature of Stack Algorithms that make these choices ideal, allowing the designer to ensure increased performance with an increase in page allocation.

Even larger systems which make use of working set principals can benefit from the trade off between approximation and full implementation. Ultimately, selecting a page replacement algorithm is not subject to any specific rule set, but is a combination of speed (comparison to Belady's Optimal Algorithm), predictability (Stack Algorithm qualifications), and hardware cost. Experimental determinations must be made for the target applications and hardware before an accurate decision can be made; in some instances the generally less-efficient algorithm will outperform a more complex implementation, simply due to an uncommon page reference stream or lack of computational overhead. It is therefore vital that a designer be conscious of the available implementations, and has sufficient data for a specific application, before deciding on a page replacement algorithm.

References

[1] G. Nutt, Operating Systems, A Modern Perspective, 2nd ed., Reading, Mass.: Addison Wesley Longman, 2000.

[2] A. Silberschatz, P. Galvin, G. Gagne, Operating Systems Concepts, 6th ed., Danvers, Mass.: John Wiley and Sons, 2003.

[3] W. Stallings, Operating Systems, Internals and Design Principals, 3rd ed., Upper Saddle River, N.J.: Prentice-Hall, 1998.

[4] C. Crowley, Operating Systems, A Design Oriented Approach, 1st ed., Chicago: Irwin, a Times Mirror Higher Education Group, 1997.

[5] D. Tsichritzis and P. Bernstein, Operating Systems, 1st ed., London: Academic Press, 1974.

[6] M. Maekawa and A. Oldehoeft, Operating Systems, Advanced Concepts, 1st ed., Menlo Park, Ca.: The Benjamin/Cummings Publishing Co., 1987.

[7] J. Feldman, Computer Architecture, A Designer's Text Based on a Generic RISC Architecture, 1st ed., New York: McGraw-Hill, 1994.

[8] G. Glass and P. Cao, “Adaptive Page Replacement Based on Memory Reference Behavior,” *Proc. Int’l Conf.* (ACM SIGMETRICS 97), University of Wisconsin-Madison, 1997.

[9] D. Hatfield, "Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance," IBM Journal of Research and Development, no. 26 Aug., pp. 58-66, 1972.

[10] P. Denning, "Virtual Memory," ACM Computing Surveys, vol. 28, no. 1 Mar., 1996.