Introduction

When you have a lot of free memory in a virtual memory manager, life is simple. When a page fault occurs, you search the free-page list for a free page and assign it to the faulting page. Congratulations, Operating System! You've done it yet again.

Unfortunately, when there isn't much RAM available, things get a little more interesting. In this circumstance, the OS is forced to start paging out pages in order to make place for actively used pages. The replacement policy of the OS determines which page (or pages) to evict; historically, it was one of the most critical decisions made by early virtual memory systems, as older systems had little physical memory. At the very least, it's an intriguing collection of policies worth learning more about. As a result, this is our issue:

How does the operating system choose which page (or pages) to remove from memory? The system's replacement policy makes this decision, which often follows some broad principles (described below) but also contains some adjustments to avoid corner-case behaviors.

Cache Management

Before we go into policy, we'll go over the problem we're seeking to tackle in greater depth. Main memory can be considered as a cache for virtual memory pages in the system because it holds a subset of all the pages in the system. As a result, our goal in selecting a replacement policy for this cache is to reduce the number of cache misses, or the number of times we must request a page from disk. Alternatively, we could think of our goal as increasing the number of cache hits, or the number of times a page is found in memory.
The average memory access time (AMAT) for a program can be calculated using the number of cache hits and misses (a metric computer architects compute for hardware caches. Specifically, we may compute the AMAT of a program using these values:

where TM denotes the cost of accessing memory, TD denotes the cost of accessing disk, and PMiss is the likelihood of the data not being found in the cache (a miss); PMiss is a number that ranges from 0.0 to 1.0, and we sometimes refer to it as a percent miss rate rather than a probability (e.g., a 10% miss rate equals PMiss = 0.10). Note that you must always pay the cost of accessing data in memory; but, if you miss, you must also pay the cost of getting data from disk.
Consider a machine with a (small) address space: 4KB, divided into 256-byte pages. As a result, a virtual address is made up of two parts: a 4-bit VPN (the most important bits) and an 8-bit offset (the least significant bits). As a result, a process in this example can access 24 or 16 virtual pages in total. The following memory references (i.e. virtual addresses) are generated in this example: 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900. These virtual addresses are the first byte of each of the address space's first ten pages (the page number being the first hex digit of each virtual address).

Assume that all pages except virtual page 3 are already stored in memory. As a result, our memory references would behave as follows: hit, hit, miss, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, The hit rate (the percentage of references found in memory) can be calculated as follows: 90%, because 9 out of 10 references are found in memory. As a result, the miss rate is 10% (PM iss = 0.1). In general, PHit + PMiss = 1.0; the hit and miss rates add up to 100%. We need to know the cost of accessing memory and the cost of accessing disk to calculate AMAT. We have the following AMAT if the cost of accessing memory (TM) is around 100 nanoseconds and the cost of accessing disk (TD) is around 10 milliseconds: 100ns + 0.1 10ms is 100ns + 1ms, or 1.0001 ms, or approximately 1 millisecond. If our hit rate had been 99.9% (Pmiss =

0.001), the outcome would have been substantially different: AMAT is 100 times faster at 10.1 microseconds. AMAT approaches 100 nanoseconds as the hit rate approaches 100%.

Unfortunately, as this example shows, the cost of disk access in modern systems is so high that even a small miss rate can quickly overwhelm the overall AMAT of running processes. We must either avoid as many misses as possible or run at the disk's speed. One method to help with this is to establish a smart policy, which we are currently doing.

The Optimal Replacement Policy

It would be useful to compare a given replacement policy to the best suitable replacement policy. The ideal replacement policy minimizes overall misses. It was demonstrated that replacing the page that would be accessed the farthest in the future is the best policy for minimizing cache misses.
Hopefully, the optimal policy's intuition makes sense. Consider this: if you must discard a page, why not the one needed the most distantly? In essence, you're suggesting that the cache's other pages are more significant than the one at the end. The rationale is simple: you will refer to other pages before the farthest out.

Let's look at a basic example to see how optimum policy generates decisions. Suppose a program reads from a stream of virtual pages like this: 0 1 2 3 Figure 22.1 depicts optimal's behavior with a three-page cache.

The figure depicts the actions. The first three accesses are misses, as the cache starts empty (or compulsory miss). Then we refer to pages 0 and 1, both cached. Finally, we reach page 3, but the cache is filled; a replacement is required! So, which page should we replace? Using the optimal strategy, we can observe that 0 gets accessible practically immediately, 1 a bit later, and 2 the furthest in the future. That leaves only pages 0, 1, and 3 in the cache, which is the ideal policy. On to page 2, which we expelled long ago, and another miss. The best policy considers the future for each cache page (0, 1, and 3), and determines that evicting page 1 (which is about to be accessed) is safe. Page 3 is evicted in the example, although 0 would also work. The trace ends on page 1.

We may also calculate the cache's hit rate: 6 hits and 5 misses is 54.5 percent. The hit rate is 85.7 percent if you disregard mandatory misses (i.e., ignore the first miss to a specific page).

Unfortunately, as with scheduling strategies, the future is unknown, hence there is no ideal strategy for a general-purpose operating system1. So, in building an actual, deployable strategy, we'll look for alternatives to deciding which pages to evict. The ideal strategy will thus only be used to gauge our progress towards "perfection".

The FIFO Policy

Many early systems used relatively simple replacement policies to avoid the complication of attempting to approach optimal. Some systems, for example, employed FIFO (first-in, first-out) replacement, in which pages were simply placed in a queue when they entered the system; when the queue was re-placed, the page at the tail of the queue (the “first-in” page) was evicted. FIFO has one major advantage: it is very easy to implement.

Let’s take a look at how FIFO performs on our hypothetical reference stream (Figure 22.2, page 5). We start our trace with three mandatory misses to pages 0, 1, and 2, before hitting on both 0 and 1. Next, page 3 is referenced, resulting in a miss; with FIFO, choosing the page that was the “first one” in (the cache state in the figure is preserved in FIFO order, with the first-in page on the left) is simple: choose page 0. Unfortunately, our next access will be to page 0, which will result in another missed opportunity and replacement (of page 1). Then we hit on page 3, but missed on 1 and 2 before eventually hitting on 3.

When compared to optimum, FIFO performs significantly worse, with a 36.4 percent hit rate (or 57.1 percent excluding compulsory misses). Even when page 0 has been viewed multiple times, FIFO still discards it because it was the first one brought into memory.

The Random Policy

Random is a similar replacement policy that simply selects a random page to replace when memory is limited. Random is similar to FIFO in that it is straightforward to construct but does not attempt to be very sophisticated in deciding which blocks to evict. Let’s check how Random performs on our well-known example reference stream (see Figure 22.3).

Of course, Random’s performance is totally dependent on how lucky (or bad) it is in its decisions. Random performs somewhat better than FIFO but slightly worse than optimum in the case above. We can repeat the Random experiment a thousand times to see how it performs in general. Figure 22.4 shows the number of hits Random gets over 10,000 trials with different random seeds. As you can see, Random performs as well as ideal at times (just over 40% of the time), achieving 6 hits on the example trace; at other times, it performs substantially worse, achieving 2 hits or less. Random’s performance is determined by the luck of the draw.

The Least Recently Used Policy

Sadly, even basic policies like FIFO or Random are likely to have one flaw: they may delete a critical page that is about to be referenced again. FIFO throws away the first page that was brought in; if this is a page with essential code or data structures on it, it will be thrown out anyway, even if it will be paged back in soon. As a result, FIFO, Random, and other similar policies are unlikely to achieve optimal results; something smarter is required.

To better our estimate at the future, we draw on the past and use history as our guide, just like we did with scheduling policy. For instance, if a software has recently viewed a page, it is likely to do so again shortly.
Frequency is one type of historical data that a page-replacement policy could employ; if a page has been accessed numerous times, it may not need to be changed because it has some value. The recency of access is a more often utilized feature of a page; the more recently a page has been accessed, the more likely it will be accessed again.

This set of principles is based on what is known as the principle of localization, which is just an observation about how programs behave. Simply put, this principle states that programs often read specific code sequences (e.g., in a loop) and data structures (e.g., an array accessed by the loop); we should try to utilize history to determine which pages are significant, and maintain those pages in memory when eviction time comes.
As a result, a family of straightforward historical-based algorithms emerges. When an eviction is required, the Least-Frequently-Used (LFU) policy replaces the least-frequently-used page. The Least-Recently-Used (LRU) policy, on the other hand, replaces the least-recently-used page. These algorithms are simple to remember: if you know the name, you'll know exactly what they do, which is a great quality in a name.

Let's look at how LRU works on our sample reference stream to get a better understanding of it. The results are shown in Figure 22.5 (page 7). The image shows how LRU can outperform stateless rules like Random or FIFO by utilizing history. When LRU first has to replace a page, it evicts page 2 because 0 and 1 have been viewed more recently. Because 1 and 3 have been accessed more recently, it replaces page 0. In both circumstances, LRU's historical judgement proves to be correct, and the subsequent references are thus hits. As a result, in our scenario, LRU performs as well as it possibly can, meeting ideal performance.

It’s also worth noting that these algorithms have polar opposites: Most-Frequently-Used (MFU) and Most-Recently-Used (MRU) (MRU). In most cases (but not all!), these approaches fail because they neglect rather than embrace the locality that most programs exhibit.

Workload Examples

Now let’s look at a few more instances to further grasp the concept. Instead of simple traces, we’ll look at complex workloads. Even these workloads are simplified; application traces would be a better study.
This means that each reference is to a random page within the set of accessible pages. In this basic example, the workload visits 10,000 pages, each one unique. The cache size is varied in the experiment from 1 page to 100 pages to investigate how each policy responds over the range of cache sizes.
Figure 22.6 shows the optimum, LRU, Random, and FIFO outcomes. The y-axis represents the hit rate for each strategy, and the x-axis shows the cache capacity.

The graph reveals a lot. First, when there is no locality in the workload, LRU, FIFO, and Random all perform equally well, with the hit rate dictated by the cache size. Second, when the cache is large enough to hold the whole workload, it doesn’t matter which policy you apply; all policies (even Random) converge to a 100% hit rate. Finally, you can show that optimal outperforms realistic policies; looking into the future, if possible, outperforms replacement.

In the “80-20” workload, 80 percent of the references are to 20 percent of the pages (the “hot” pages), and 20 percent are to the remaining 80 percent of the pages (the “cold” pages). Our task has 100 unique pages, therefore “hot” pages are usually referred to, and “cold” pages are the rest. (Page 10) demonstrates the policies’ performance on this workload.

As seen in the image, while both random and FIFO do well, LRU does better since it is more likely to keep the hot pages, which are likely to be referred to again shortly. Optimal outperforms LRU again, proving that past data is not ideal.

So, is LRU’s improvement over Random and FIFO significant? The common answer is “it depends.” If each miss is costly, even a little increase in hit rate (down in miss rate) can significantly improve performance. If misses are less costly, the benefits of LRU are less essential.
Let’s look at one last task. There are 10,000 unique page accesses in this workload. It is called the “looping sequential” task because it refers to 50 pages in succession starting at 0, then 1,…, up to page 49. Figure 22.8 depicts the policies’ behavior under this workload.

For both LRU and FIFO, this workload is a worst-case scenario in many applications (including commercial applications like databases). Due to the cyclical nature of the workload, these earlier pages will be accessed sooner than the pages that the policies desire to preserve in cache. Even with a 49-page cache, a 50-page looping-sequential workload results in a 0% hit rate. Random, on the other hand, does far better, obtaining a non-zero hit rate. Random has some nice qualities, such not having odd corner-case behaviors.

Historical Algorithm Implementation

As you can see, algorithms like LRU outperform simpler rules like FIFO or Random, which may miss essential pages. Sadly, previous policies provide a fresh problem: how to execute them?

Consider LRU. We need to work hard to polish it. We must update some data structure to advance this page to the front of the list for each page access (i.e., memory access) (i.e., the MRU side). FIFO, on the other hand, only accesses the list of pages when a page is evicted (by removing the first-in page) or added (to the last-in side). Every memory reference must be accounted for in order to maintain track of which pages are the least and most recently used. Such accounting might clearly harm performance if not done properly.

Adding some hardware support could help speed things up. So a machine might update a time field in memory on each page access (for example, this could be in the per-process page table, or just in some separate array in memory, with one entry per physical page of the system). So, when a page is accessed, the time field is set by hardware. The OS may then scan all time fields to find the least-used page and replace it.

Unfortunately, as a system's page count grows, scanning a wide array of times to discover the least-used page becomes prohibitively expensive. So, 4GB RAM divided into 4KB pages. With 1 million pages, even a contemporary CPU will take a long time to find the LRU page. So, do we need to discover the oldest page to replace? Can we live with approximations?

Approximating LRU

The answer is yes: approximating LRU reduces computing overhead and is used by many modern systems. The first system with paging, the Atlas one-level store, used a use bit (sometimes called a reference bit) to support the idea. Each system page has one usage bit, which is stored in memory (they could be in the per-process page tables, for example, or just in an array somewhere). When a page is referenced (read or written), the hardware sets the use bit to 1. The OS is responsible for clearing the bit (setting it to 0).

Use bit to approximate LRU by OS? Well, there could be many ways, but the clock algorithm offered one straightforward option. Imagine the system's pages organized in a circle. Initially, a clock hand points to any page. When replacing a page, the OS checks if the use bit is 1 or 0. If 1, page P was recently used and so is not a strong replacement choice. The usage bit for P is cleared, and the clock hand is advanced one page (P + 1). If the use bit is set to 0, the page hasn't been used recently (or, in the worst case, that all pages have been and that we have now searched through the entire set of pages, clearing all the bits).

This is not the only way to approximate LRU with a usage bit. Any method that periodically clears the use bits and then compares which pages have 1 vs 0 use bits to decide which to replace is fine. Corbato's clock method was an early success, as it avoided continually checking memory for unoccupied pages.

Figure 22.9 depicts a clock algorithm variant's behavior. When doing a replacement, this version randomly checks pages for reference bits, clearing them when they are set to 1 and selecting them as victims when they are set to 0. As you can see, it outperforms perfect LRU and techniques that ignore history.

Dirty Pages

The addition of a consideration of whether a page has been updated or not while in memory is a simple improvement to the clock method that is routinely made. The reason for this is that if a page has been updated and is thus unclean, it must be evicted by writing it back to disk, which is costly. The eviction is free if it has not been updated (and so is clean); the physical frame can simply be reused for other purposes without further I/O. As a result, certain virtual machine systems prefer to evict clean pages over unclean ones.

A modified bit should be included in the hardware to accommodate this behavior (a.k.a. dirty bit). Because this bit is set whenever a page is written, it can be used in the page-replacement method. For example, the clock algorithm may be altered to look for pages that are both unused and clean to evict first; if those aren't found, then look for unused dirty pages, and so on.

Other VM Policies

The VM subsystem’s page replacement policy isn’t the only one it implements (though it may be the most important). The operating system, for example, must decide when to load a page into memory. This policy, often known as the page selection policy, provides the OS with a number of alternatives.
The OS simply employs demand paging for most pages, which means the OS loads the page into memory as needed. Of course, the OS may predict when a page will be used and bring it in ahead of time; this is known as prefetching, and it should only be done when there is a good likelihood of success. Some systems, for example, will presume that if code page P is brought into memory, code page P +1 will be accessed shortly and so should be brought into memory as well.

Another policy controls how the operating system writes pages to disk. Of course, they may be written out one at a time; however, many systems instead pool many pending writes in memory and write them to disk in a single (more efficient) write. This is known as clustering or simply grouping of writes, and it works because of the nature of disk drives, which can perform a single large write more efficiently than a series of little ones.

Thrashing

Before we wrap up, we'll address one last question: what should the OS do when memory is simply overcrowded, and the memory demands of the operating processes simply outnumber the physical memory available? In this situation, the system will be continuously paging, a condition known as thrashing.
Some early operating systems have a pretty complex collection of mechanisms for both detecting and dealing with thrashing. For example, given a set of processes, a system might opt not to execute a subset of them in the hopes that the working sets (the pages that they are currently using) of the reduced set of processes will fit in memory and therefore allow progress. This method, known as admission control, asserts that it is sometimes better to accomplish less work properly than to try to complete everything poorly all at once, a circumstance that we frequently find in real life and in modern computer systems (sadly).

Some modern systems take a harsher stance against memory saturation. When memory is oversubscribed, some versions of Linux, for example, launch an out-of-memory killer; this daemon selects a memory-intensive process and kills it, decreasing memory in a not-so-subtle manner. While this strategy is effective at decreasing memory strain, it might cause difficulties if it, for example, disables the X server, rendering any apps that require the display inoperable.

Summary

We’ve seen the introduction of various page-replacement (and other) rules, which are now part of every modern operating system’s VM subsystem. Recent systems make some adjustments to simple LRU approximations like clock; for example, scan resistance is a major feature of many modern algorithms like ARC. Scan-resistant algorithms are typically LRU-like, but they also attempt to avoid the worst-case behavior of LRU, which we observed with the looping-sequential workload. As a result, the advancement of page-replacement algorithms continues.

However, the importance of such techniques has decreased in many circumstances as the difference between memory-access and disk-access durations has increased. Because paging to disk is so expensive, the expense of frequent paging is prohibitive. As a result, the greatest remedy to excessive paging is frequently a simple (albeit intellectually disappointing) one: buy additional memory.