21

Beyond Physical Memory: Policies

In a virtual memory manager, life is easy when you have a lot of free memory. A page fault occurs, you find a free page on the free-page list, and assign it to the faulting page. Hey, Operating System, congratulations! You did it again.

Unfortunately, things get a little more interesting when little memory is free. In such a case, this **memory pressure** forces the OS to start **paging out** pages to make room for actively-used pages. Deciding which page (or pages) to **evict** is encapsulated within the **replacement policy** of the OS; historically, it was one of the most important decisions the early virtual memory systems made, as older systems had little physical memory. Minimally, it is an interesting set of policies worth knowing a little more about. And thus our problem:

THE CRUX: HOW TO DECIDE WHICH PAGE TO EVICT
How can the OS decide which page (or pages) to evict from memory? This decision is made by the replacement policy of the system, which usually follows some general principles (discussed below) but also includes certain tweaks to avoid corner-case behaviors.

21.1 Cache Management

Before diving into policies, we first describe the problem we are trying to solve in more detail. Given that main memory holds some subset of all the pages in the system, it can rightly be viewed as a **cache** for virtual memory pages in the system. Thus, our goal in picking a replacement policy for this cache is to minimize the number of **cache misses**; that is, to minimize the number of times that we have to go to disk to fetch the desired page. Alternately, one can view our goal as maximizing the number of **cache hits**, the number of times a page that is accessed is found in memory.

Knowing the number of cache hits and misses let us calculate the **average memory access time** (**AMAT**) for a program (a metric computer architects compute for hardware caches [HP06]). Specifically, given these values, we can compute the AMAT of a program as follows: $(Hit_{\%} \cdot T_M) + (Miss_{\%} \cdot T_D)$, where T_M is the cost of accessing memory, and T_D the cost of accessing disk.

For example, let us imagine a machine with a (tiny) address space: 4KB, with 256-byte pages. Thus, a virtual address has two components: a 4-bit VPN (the most-significant bits) and an 8-bit offset (the least-significant bits). Thus, a process in this example can access 2^4 or 16 total virtual pages. In this example, the process generates the following memory references (i.e., virtual addresses): 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900. These virtual addresses refer to the first byte of each of the first ten pages of the address space (the page number being the first hex digit of each virtual address).

Let us further assume that every page except virtual page 3 are already in memory. Thus, our sequence of memory references will encounter the following behavior: hit, hit, hit, miss, hit, hit, hit, hit, hit, hit. We can compute the **hit rate** (the percent of references found in memory): 90%, as 9 out of 10 references are in memory. The **miss rate** is obviously 10%.

To calculate AMAT, we simply need to know the cost of accessing memory and the cost of accessing disk. Assuming the cost of accessing memory (T_M) is around 100 nanoseconds, and the cost of accessing disk (T_D) is about 10 milliseconds, we have the following AMAT: $0.9 \cdot 100ns + 0.1 \cdot 10ms$, which is $90ns + 1ms$, or 1.00009 ms, or about 1 millisecond. If our hit rate had instead been 99.9%, the result is quite different: AMAT is 10.1 microseconds, or roughly 100

times faster. As the hit rate approaches 100%, AMAT approaches 100 nanoseconds.

Unfortunately, as you can see in this example, the cost of disk access is so high in modern systems that even a tiny miss rate will quickly dominate the overall AMAT of running programs. Clearly, we need to avoid as many misses as possible or run slowly, at the rate of the disk. One way to help with this is to carefully develop a smart policy, as we now do.

21.2 The Optimal Replacement Policy

To better understand how a particular replacement policy works, it would be nice to compare it to the best possible replacement policy. As it turns out, such an **optimal** policy was developed by Belady many years ago [B66] (he originally called it MIN). The optimal replacement policy leads to the fewest number of misses overall. Belady showed that a simple (but, unfortunately, difficult to implement!) approach that replaces the page that will be accessed *furthest in the future* is the optimal policy, resulting in the fewest-possible cache misses.

Hopefully, the intuition behind the optimal policy makes sense. Think about it like this: if you have to throw out some page, why not throw out the one that is needed the furthest from now? By doing so, you are essentially saying that all the other pages in the cache are more important than the one furthest out. The reason this is true is simple: you will refer to the other pages before you refer to the one furthest out.

Let's trace through a simple example to understand the decisions the optimal policy makes. Assume a program accesses the following stream of virtual pages: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1. Table 21.1 shows what the optimal policy would do for this reference stream, assuming a cache that fits only three pages.

In the table, you can see the following actions. Not surprisingly, the first three accesses are misses, as the cache begins in an empty state; such a miss is sometimes referred to as a **cold-start miss** (or **compulsory miss**). Then we refer again to pages 0 and 1, which both hit in the cache. Finally, we reach another miss (to page 3), but this time the cache is full; a replacement must take place! Which begs the question: which page should we replace? With the optimal policy,

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Table 21.1: Tracing the Optimal Policy

we examine the future for each page currently in the cache (0, 1, and 2), and see that 0 is accessed almost immediately, 1 is accessed a little later, and 2 is accessed furthest in the future. Thus the optimal policy has an easy choice: evict page 2, resulting in pages 0, 1, and 3 in the cache. The next three references are hits, but then we get to page 2, which we evicted long ago, and suffer another miss. Here the optimal policy again examines the future for each page in the cache (0, 1, and 3), and sees that as long as it doesn't evict page 1 (which is about to be accessed), we'll be OK. The example shows page 3 getting evicted, although 0 would have been a fine choice too. Finally, we hit on page 1 and the trace completes.

We can also calculate a hit rate for the cache given this stream. With 6 hits and 5 misses, the hit rate is $\frac{Hits}{Hits+Misses}$ which is $\frac{6}{6+5}$ or 54.6%. You can also compute the hit rate *modulo* compulsory misses (i.e., ignore the *first* miss to any given page), resulting in a more impressive 85.7% hit rate.

Unfortunately, as we saw before in the development of scheduling policies, the future is not generally known; you can't build the optimal policy for a general-purpose operating system[1]. Thus, in developing a real, deployable policy, we will have to focus on approaches that find some other way to decide which page to evict. The optimal policy will thus serve only as a comparison point, to know how close we are to "perfect".

[1]If you can, let us know! We can become rich together. Or, like the scientists who "discovered" cold fusion, widely scorned and mocked.

ASIDE: TYPES OF CACHE MISSES

In the computer architecture world, architects sometimes find it useful to characterize misses by type, into one of three categories: compulsory, capacity, and conflict misses, sometimes called the **Three C's** [H87]. A **compulsory miss** (or **cold-start miss** [EF78]) occurs because the cache is empty to begin with and this is the first reference to the item; in contrast, a **capacity miss** occurs because the cache ran out of space and had to evict an item to bring a new item into the cache. The third type of miss (a **conflict miss**) arises in hardware because of limits on where an item can be placed in a hardware cache, due to something known as **set-associativity**; it does not arise in the OS page cache because such caches are always **fully-associative**, i.e., there are no restrictions on where in memory a page can be placed. See H&P for details [HP06].

21.3 A Simple Policy: FIFO

Many early systems avoided the complexity of trying to approach optimal and employed very simple replacement policies. For example, some systems used **FIFO** (first-in, first-out) replacement, where pages were simply placed in a queue when they enter the system; when a replacement occurs, the page on the tail of the queue (the "first-in" page) is evicted. FIFO has one great strength: it is quite simple to implement.

Let's examine how FIFO does on our example reference stream from above. Table 21.2 shows the results. We again begin our trace with three compulsory misses to pages 0, 1, and 2, and then hit on both 0 and 1. Next, page 3 is referenced, causing a miss; the replacement decision is easy with FIFO: pick the page that was the first-one in (the cache state in the table is kept in FIFO order, with the first-in page on the left), which is page 0. Unfortunately, our next access is to page 0(!), thus causing another miss, and another replacement (of page 1). We then hit on page 3, but miss on 1 and 2, and finally hit on 3 to finish.

Comparing FIFO to optimal, FIFO does notably worse: a 36.4% hit rate (or 57.1% excluding compulsory misses). FIFO simply can't determine the importance of blocks: even though page 0 had been accessed a number of times, FIFO still kicks it out, simply because it was the first one brought into memory.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		First-in→ 0
1	Miss		First-in→ 0, 1
2	Miss		First-in→ 0, 1, 2
0	Hit		First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2
3	Miss	0	First-in→ 1, 2, 3
0	Miss	1	First-in→ 2, 3, 0
3	Hit		First-in→ 2, 3, 0
1	Miss	2	First-in→ 3, 0, 1
2	Miss	3	First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2

Table 21.2: Tracing the FIFO Policy

21.4 Another Simple Policy: Random

Another similar replacement policy is Random, which simply picks a random page to replace under memory pressure. Random has properties similar to FIFO; it is simple to implement, but it doesn't really try to be too intelligent in picking which blocks to evict. Let's look at how Random does on our famous example reference stream (see Table 21.3).

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Table 21.3: Tracing the Random Policy

Of course, how Random does depends entirely upon how lucky (or unlucky) Random gets in its choices. In the example above, Random does a little better than FIFO, and a little worse than optimal. In fact, we can run the Random experiment thousands of times and determine how it does in general. Figure 21.1 shows how many hits Random achieves over 10,000 trials, each with a different random seed. As you can see, sometimes (just over 40% of the time), Random

is as good as optimal, achieving 6 hits on the example trace; sometimes it does much worse, achieving 2 hits or fewer. How Random does depends on the luck of the draw.

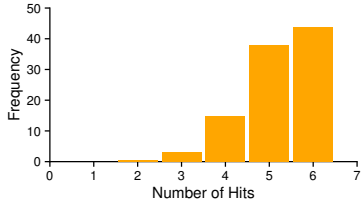

Figure 21.1: Random Performance over 10,000 Trials

ASIDE: TYPES OF LOCALITY

There are two types of locality that programs tend to exhibit. The first is known as **spatial locality**, which states that if a page P is accessed, it is likely the pages around it (say $P-1$ or $P+1$) will also likely be accessed. The second is **temporal locality**, which states that pages that have been accessed in the near past are likely to be accessed again in the near future. The assumption of the presence of these types of locality plays a large role in the caching hierarchies of hardware systems, which deploy many levels of instruction, data, and address-translation caching to help programs run fast when such locality exists.

Of course, the **principle of locality**, as it is often called, is no hard-and-fast rule that all programs must obey. Indeed, some programs access memory (or disk) in rather random fashion and don't exhibit much or any locality in their access streams. Thus, while locality is a good thing to keep in mind while designing caches of any kind (hardware or software), it does not *guarantee* success. Rather, it is a heuristic that often proves useful in the design of computer systems.

21.5 Using History: LRU

Unfortunately, any policy as simple as FIFO or Random is likely to have a common problem: it might kick out an important page, one that is about to be referenced again. FIFO kicks out the page that was first brought in; if this happens to be a page with important code or data structures upon it, it gets thrown out anyhow, even though it will soon be paged back in. Thus, FIFO, Random, and similar policies are not likely to approach optimal; something smarter is needed.

As we did with scheduling policy, to improve our guess at the future, we once again lean on the past and use *history* as our guide. For example, if a program has accessed a page in the near past, it is likely to access it again in the near future.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Table 21.4: Tracing the LRU Policy

One type of historical information a page-replacement policy could use is **frequency**; if a page has been accessed many times, perhaps it should not be replaced as it clearly has some value. An even more commonly-used property of a page is its **recency** of access; the more recently a page has been accessed, perhaps the more likely it will be accessed again.

This family of policies is based on what people refer to as the **principle of locality** [D70], which basically is just an observation about programs and their behavior. What this principle says, quite simply, is that programs tend to access certain code sequences and data structures quite frequently; we should thus try to use history to figure out which pages are important, and keep those pages in memory when it comes to eviction time.

And thus, a family of simple historically-based algorithms are born. The **Least-Frequently-Used** (**LFU**) policy replaces the least-

ASIDE: BELADY'S ANOMALY

Some year's ago, Belady (of the optimal policy) and colleagues found an interesting reference stream that behaved a little unexpectedly [BNS69]. The memory-reference stream: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. The replacement policy they were studying was FIFO. And now, the interesting part: how the cache hit rate changed when moving from a cache size of 3 to 4 pages.

In general, you would expect the cache hit rate to *increase* (get better) when the cache gets larger, right? But in this case, when running FIFO, it turns out that with a cache of size 3, 3 hits (9 misses) take place, but with a larger cache of size 4, only 2 hits (10 misses) occur. If you don't believe it, calculate the hits and misses yourself! This odd behavior is generally referred to as **Belady's Anomaly** (to the chagrin of his co-authors).

Some other policies, such as LRU, don't suffer from this problem. Can you guess why? As it turns out, LRU (and some other policies) have what is known as a **stack property** [M+70]. For algorithms that have the stack property, a cache of size $N + 1$ naturally includes the contents of a cache of size N which uses the same replacement algorithm. Thus, when increasing the cache size, you will never see a decrease in hit rate – only an increase or at worst the same hit rate. FIFO and Random (among others) clearly do not have a stack property, and thus are susceptible to anomalous behavior.

frequently-used page when an eviction must take place. Similarly, the **Least-Recently-Used** (**LRU**) policy replaces the least-recently-used page. These algorithms are easy to remember: once you know the name, you know exactly what it does.

To better understand this, let's examine how LRU does on our example reference stream. Table 21.4 shows the results. From the table, you can see how LRU can use history to do better than stateless policies such as Random or FIFO. In the example, LRU evicts page 2 when it first has to replace a page, because 0 and 1 have been accessed more recently. It then replaces page 0 because 1 and 3 have been accessed more recently. In both cases, LRU's decision, based on history, turns out to be correct, and the next references are thus hits. Thus, in our simple example, LRU does as well as possible, matching optimal in its performance.

We should also note that the opposites of these algorithms exist: **Most-Frequently-Used** (**MFU**) and **Most-Recently-Used** (**MRU**). However, in most cases (though not all!), these policies do not work well, as they ignore the locality most programs exhibit instead of embracing it.

21.6 Workload Examples

Let's look at a few more examples in order to better understand how some of these policies behave. We'll look at more complex **workloads** instead just a small trace of references.

Our first workload has no locality, which means that each reference is to a random page within the set of accessed pages. In this simple example, the workload accesses 100 unique pages over time, choosing the next page to refer to at random; overall, 10,000 pages are accessed. In the experiment, we vary the cache size from very small (1 page) to enough to hold all the unique pages (100 page), in order to see how each policy behaves over the range of cache sizes.

Figure 21.2 plots the results of the experiment for the optimal, LRU, Random, and FIFO policies. The y-axis of the figure shows the hit rate that each policy achieves; the x-axis varies the cache size as described above.

We can draw a number of conclusions from the graph. First, when there is no locality in the workload, it doesn't matter much which realistic policy you are using; LRU, FIFO, and Random all perform the same, with the hit rate exactly determined by the size of the cache. Second, when the cache is large enough to fit the entire workload, it also doesn't matter which policy you use; all policies (even optimal) converge to a 100% hit rate when all the referenced blocks fit in cache. Finally, you can see that optimal performs noticeably better than the realistic policies; peeking into the future, if it were possible, does a much better job of replacement.

The next workload we examine is called the "80-20" workload, because it has locality within it. Specifically, 80% of the references are made to 20% of the pages (you might call these pages "hot"); the remaining 20% of the references are made to the remaining 80% of the pages (perhaps these are the "cold" pages). In our workload, there are a total 100 unique pages again; thus, "hot" pages (0–19) are referred to most of the time, and "cold" pages (20-99) the remain-

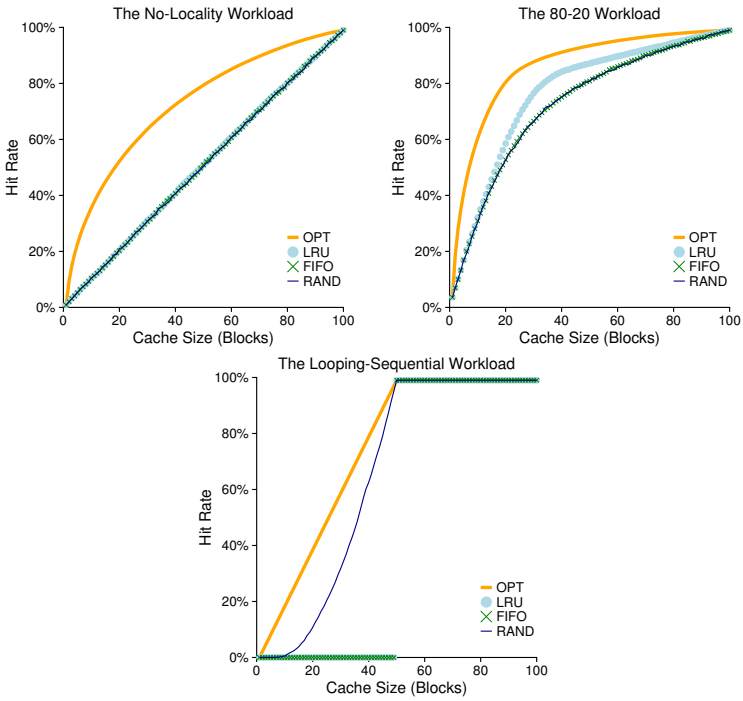

Figure 21.2: The No-Locality, 80-20, and Looping Workloads

der. Figure 21.2 also shows how the policies perform with the 80-20 workload.

As you can see from the figure, while both random and FIFO do reasonably well, LRU does better, as it is more likely to hold onto the hot pages; as those pages have been referred to frequently in the past, they are likely to be referred to again in the near future. Optimal once again does better, showing that LRU's historical information is not perfect.

You might now be wondering: is LRU's improvement over Random and FIFO really that big of a deal? The answer, as usual, is "it depends." If each miss is very costly (not uncommon), then even a

small increase in hit rate (reduction in miss rate) can make a huge difference on performance. If misses are not so costly, then of course the benefits possible with LRU are not nearly as important.

Let's look at one final workload. We call this one the "looping sequential" workload, as in it, we refer to 50 pages in sequence, starting at 0, then 1, ..., up to page 49, and then we loop, repeating those accesses, for a total of 10,000 accesses to 50 unique pages. The last graph in Figure 21.2 shows the behavior of the policies under this workload.

This workload, common in many applications (including important commercial applications such as databases [CD85]), represents a worst-case for both LRU and FIFO. These algorithms, under a looping-sequential workload, kick out older pages; unfortunately, due to the looping nature of the workload, these older pages are going to be accessed sooner than the pages that the policies prefer to keep in cache. Indeed, even with a cache of size 49, a looping-sequential workload of 50 pages results in a 0% hit rate. Interestingly, Random fares notably better, not quite approaching optimal, but at least achieving a non-zero hit rate.

DESIGN TIP: COMPARING AGAINST OPTIMAL IS USEFUL
Although optimal is not very practical as a real policy, it is incredibly useful as a comparison point in simulation or other studies. Saying that your fancy new algorithm has a 80% hit rate isn't meaningful in isolation; saying that optimal achieves an 82% hit rate (and thus your new approach is quite close to optimal) makes the result more meaningful and gives it context. Thus, in any study you perform, knowing what the optimal is lets you perform a better comparison, showing how much improvement is still possible, and also when you can *stop* making your policy better, because it is close enough to the ideal [AD03].

21.7 Implementing Historical Algorithms

As you can see, an algorithm such as LRU can generally do a better job than simpler policies like FIFO or Random, which may throw out important pages. Unfortunately, historical policies present us with a new challenge: how do we implement them?

Let's take, for example, LRU. To implement it perfectly, we need to do a lot of work. Specifically, upon each *page access* (i.e., each memory access, whether an instruction fetch or a load or store), we must update some data structure to move this page to the front of the list (i.e., the MRU side). Contrast this to FIFO, where the FIFO list of pages is only accessed when a page is evicted (by removing the first-in page) or when a new page is added to the list (to the last-in side). To keep track of which pages have been least- and most-recently used, the system has to do some accounting work *on every memory reference.* Clearly, without great care, such accounting could greatly reduce performance.

One method that could help speed this up is to add a little bit of hardware support. For example, a machine could update, on each page access, a time field in memory (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). Thus, when a page is accessed, the time field would be set, by hardware, to the current time. Then, when replacing a page, the OS could simply scan all the time fields in the system to find the least-recently-used page.

Unfortunately, as the number of pages in a system grows, scanning a huge array of times just to find the absolute least-recently-used page is prohibitively expensive. Imagine a modern machine with 4GB of memory, chopped into 4KB pages. This machine has 1 million pages, and thus finding the LRU page will take a long time, even at modern CPU speeds. Which begs the question: do we really need to find the absolute oldest page to replace? Can we instead survive with an approximation?

CRUX: HOW TO IMPLEMENT AN LRU REPLACEMENT POLICY
Given that it will be expensive to implement perfect LRU, can we approximate it in some way, and still obtain the desired behavior?

21.8 Approximating LRU

As it turns out, the answer is yes: approximating LRU is more feasible from a computational-overhead standpoint, and indeed it is what many modern systems do. The idea requires some hardware

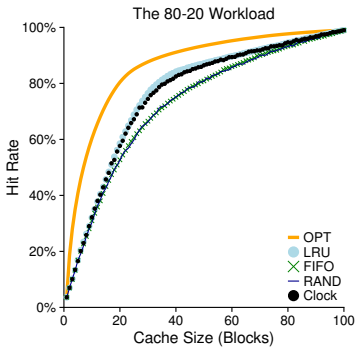

Figure 21.3: The 80-20 Workload with Clock

support, in the form of a **use bit** (sometimes called the **reference bit**), the first of which was implemented in the first system with paging, the Atlas one-level store [KE+62]. There is one use bit per page of the system, and the use bits live in memory somewhere (they could be in the per-process page tables, for example, or just in an array somewhere). Whenever a page is referenced (i.e., read or written), the use bit is set by hardware to 1. The hardware never clears the bit, though (i.e., sets it to 0); that is the responsibility of the OS.

How does the OS employ the use bit to approximate LRU? Well, there could be a lot of ways, but with the **clock algorithm** [C69], one simple approach was suggested. Imagine all the pages of the system arranged in a circular list. A **clock hand** points to some particular page to begin with (it doesn't really matter which). When a replacement must occur, the OS checks if the currently-pointed to page P has a use bit of 1 or 0. If 1, this implies that page P was recently used and thus is *not* a good candidate for replacement. Thus, the clock hand is incremented to the next page $P + 1$, and the use bit for P set to 0 (cleared). The algorithm continues until it finds a use bit that is set to 0, implying this page has not been recently used (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).

Note that this approach is not the only way to employ a use bit to approximate LRU. Indeed, any approach which periodically clears the use bits and then differentiates between which pages have use

bits of 1 versus 0 to decide which to replace would be fine. The clock algorithm of Corbato's was just one early approach which met with some success, and had the nice property of not repeatedly scanning through all of memory looking for an unused page.

The behavior of a clock algorithm variant is shown in Figure 21.3. This variant randomly scans pages when doing a replacement; when it encounters a page with a reference bit set to 1, it clears the bit (i.e., sets it to 0); when it finds a page with the reference bit set to 0, it chooses it as its victim. As you can see, although it doesn't do quite as well as perfect LRU, it does better than approaches that don't consider history at all.

21.9 Considering Dirty Pages

One small modification to the clock algorithm (also originally suggested by Corbato [C69]) that is commonly made is the additional consideration of whether a page has been modified or not while in memory. The reason for this: if a page has been **modified** and is thus **dirty**, it must be written back to disk to evict it, which is expensive. If it has not been modified (and is thus **clean**), the eviction is free; the physical frame can simply be reused for other purposes without additional I/O. Thus, some VM systems prefer to evict clean pages over dirty pages.

To support this behavior, the hardware should include a **modified bit** (a.k.a. **dirty bit**). This bit is set any time a page is written, and thus can be incorporated into the page-replacement algorithm. The clock algorithm, for example, could be changed to scan for pages that are both unused and clean to evict first; failing to find those, then for unused pages that are dirty; etc.

21.10 Other VM Policies

Page replacement is not the only policy the VM subsystem employs (though it may be the most important). For example, the OS also has to decide *when* to bring a page into memory. This policy, sometimes called the **page selection** policy [D70], presents the OS with some different options.

For most pages, the OS simply uses **demand paging**, which means the OS brings the page into memory when it is accessed, "on de-

mand" as it were. Of course, the OS could guess that a page is about to be used, and thus bring it in ahead of time; this behavior is known as **prefetching** and should only be done when there is reasonable chance of success. For example, some systems will assume that if a code page P is brought into memory, that code page $P+1$ will likely soon be accessed and thus should be brought into memory too.

Another policy determines how the OS writes pages out to disk. Of course, they could simply be written out one at a time; however, many systems instead collect a number of pending writes together in memory and write them to disk in one (more efficient) write. This behavior is usually called **clustering** or simply **grouping** of writes, and is effective because of the nature of disk drives, which perform a single, large write more efficiently than many small writes (as we will see).

21.11 Thrashing

Before closing, we address one final question: what should the OS do when memory is simply oversubscribed, and the memory demands of the set of running processes simply exceeds the available physical memory? In this case, the system will constantly be paging, a condition sometimes referred to as **thrashing** [D70].

Some earlier operating systems had a fairly sophisticated set of mechanisms to both detect and cope with thrashing when it took place. For example, given a set of processes, a system could decide not to run a subset of processes, with the hope that the reduced set of processes **working sets** (the pages that they are using actively) fit in memory and thus can make progress. This approach, generally known as **admission control**, states that it is sometimes better to do less work well than to try to do everything at once and make little or no progress in all directions, a situation we often encounter in real life as well.

More modern systems sometimes take more a draconian approach. For example, some versions of Linux run an "out-of-memory killer" when memory is oversubscribed; this daemon picks a random process and kills it, thus reducing memory in a not-too-subtle manner. While successful at reducing memory pressure, this approach has its problem, if, for example, it kills the X server and thus renders any applications that use the display unusable.

21.12 Summary

We have seen the introduction of a number of page-replacement (and other) policies, which are part of the VM subsystem of all modern operating systems. Modern systems add some tweaks to straightforward LRU approximations like clock; for example, **scan resistance** is an important part of many modern algorithms, such as ARC [MM03]. Scan-resistant algorithms are usually LRU-like but also try to avoid the worst-case behavior of LRU, which we saw with the looping-sequential workload. Thus, the evolution of page-replacement algorithms continues.

However, in many cases the importance of said algorithms has decreased, as the discrepancy between memory-access and disk-access times has increased. Because paging to disk is so expensive, the cost of frequent paging is prohibitive. Thus, the best solution to excessive paging is often a simple (if intellectually dissatisfying) one: buy more memory.

References

[AD03] "Run-Time Adaptation in River"
Remzi H. Arpaci-Dusseau
ACM TOCS, 21:1, February 2003
A summary of one of the authors' dissertation work on a system named River. Certainly one place where he learned that comparison against the ideal is an important technique for system designers.

[B66] "A Study of Replacement Algorithms for Virtual-Storage Computer"
Laszlo A. Belady
IBM Systems Journal 5(2): 78-101, 1966
The paper that introduces the simple way to compute the optimal behavior of a policy (the MIN algorithm).

[BNS69] "An anomaly in space-time characteristics of certain programs running in a paging machine"
L. A. Belady and R. A. Nelson and G. S. Shedler
Communications of the ACM, 12:6, June 1969
Introduction of the little sequence of memory references known as Belady's Anomaly. How do Nelson and Shedler feel about this name, we wonder?

[CD85] "An evaluation of buffer management strategies for relational database systems"
Hong-Tai Chou and David J. DeWitt
VLDB '85, Stockholm, Sweden, August 1985
A famous database paper on the different buffering strategies you should use under a number of common database access patterns.

[C69] "A paging experiment with the Multics system"
F.J. Corbato
Included in a Festschrift published in honor of Prof. P.M. Morse
MIT Press, Cambridge, MA, 1969
The original (and hard to find!) reference to the clock algorithm, though not the first usage of a use bit. Thanks to H. Balakrishnan of MIT for digging up this paper for us.

[D70] "Virtual Memory"
Peter J. Denning
Computing Surveys, Vol. 2, No. 3, September 1970
Denning's early and famous survey on virtual memory systems.

[EF78] "Cold-start vs. Warm-start Miss Ratios"
Malcolm C. Easton and Ronald Fagin
Communications of the ACM, 21:10, October 1978
A good discussion of cold-start vs. warm-start misses.

[HP06] "Computer Architecture: A Quantitative Approach"
John Hennessy and David Patterson
Morgan-Kaufmann, 2006
A great and marvelous book about computer architecture. Read it!

[H87] "Aspects of Cache Memory and Instruction Buffer Performance"
Mark D. Hill
Ph.D. Dissertation, U.C. Berkeley, 1987
Mark Hill, in his dissertation work, introduced the Three C's, which later gained wide popularity with its inclusion in H&P [HP06]. The quote from therein: "I have found it useful to partition misses ... into three components intuitively based on the cause of the misses (page 49)."

[KE+62] "One-level Storage System"
T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner
IRE Trans. EC-11:2, 1962
Although Atlas had a use bit, it only had a very small number of pages, and thus the scanning of the use bits in large memories was not a problem the authors solved.

[M+70] "Evaluation techniques for storage hierarchies"
R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger
IBM Systems Journal, Volume 9:2, 1970
A paper that is mostly about how to simulate cache hierarchies efficiently; certainly a classic in that regard, as well for its excellent discussion of some of the properties of various replacement algorithms. Can you figure out why the stack property might be useful for simulating a lot of different-sized caches at once?

[MM03] "ARC: A Self-Tuning, Low Overhead Replacement Cache"
Nimrod Megiddo and Dharmendra S. Modha
FAST 2003, February 2003, San Jose, California
An excellent modern paper about replacement algorithms, which includes a new policy, ARC, that is now used in some systems.

Homework

This simulator, `paging-policy.py`, allows you to play around with different page-replacement policies. For example, let's examine how LRU performs with a series of page references with a cache of size 3:

0 1 2 0 1 3 0 3 1 2 1

To do so, run the simulator as follows:

```
prompt> ./paging-policy.py --addresses=0,1,2,0,1,3,0,3,1,2,1
                           --policy=LRU --cachesize=3 -c
```

And what you would see is:

```
ARG addresses 0,1,2,0,1,3,0,3,1,2,1
ARG numaddrs 10
ARG policy LRU
ARG cachesize 3
ARG maxpage 10
ARG seed 0

Solving...

Access: 0 MISS LRU->      [0]<-MRU Replace:- [Hits:0 Misses:1]
Access: 1 MISS LRU->   [0, 1]<-MRU Replace:- [Hits:0 Misses:2]
Access: 2 MISS LRU->[0, 1, 2]<-MRU Replace:- [Hits:0 Misses:3]
Access: 0 HIT  LRU->[1, 2, 0]<-MRU Replace:- [Hits:1 Misses:3]
Access: 1 HIT  LRU->[2, 0, 1]<-MRU Replace:- [Hits:2 Misses:3]
Access: 3 MISS LRU->[0, 1, 3]<-MRU Replace:2 [Hits:2 Misses:4]
Access: 0 HIT  LRU->[1, 3, 0]<-MRU Replace:2 [Hits:3 Misses:4]
Access: 3 HIT  LRU->[1, 0, 3]<-MRU Replace:2 [Hits:4 Misses:4]
Access: 1 HIT  LRU->[0, 3, 1]<-MRU Replace:2 [Hits:5 Misses:4]
Access: 2 MISS LRU->[3, 1, 2]<-MRU Replace:0 [Hits:5 Misses:5]
Access: 1 HIT  LRU->[3, 2, 1]<-MRU Replace:0 [Hits:6 Misses:5]
```

The complete set of possible arguments for paging-policy is listed on the following page, and includes a number of options for varying the policy, how addresses are specified/generated, and other important parameters such as the size of the cache.

```
prompt> ./paging-policy.py --help
Usage: paging-policy.py [options]

Options:
-h, --help      show this help message and exit
-a ADDRESSES, --addresses=ADDRESSES
                a set of comma-separated pages to access;
                -1 means randomly generate
-f ADDRESSFILE, --addressfile=ADDRESSFILE
                a file with a bunch of addresses in it
-n NUMADDRS, --numaddrs=NUMADDRS
                if -a (--addresses) is -1, this is the
                number of addrs to generate
-p POLICY, --policy=POLICY
                replacement policy: FIFO, LRU, LFU, OPT,
                                    UNOPT, RAND, CLOCK
-b CLOCKBITS, --clockbits=CLOCKBITS
                for CLOCK policy, how many clock bits to use
-C CACHESIZE, --cachesize=CACHESIZE
                size of the page cache, in pages
-m MAXPAGE, --maxpage=MAXPAGE
                if randomly generating page accesses,
                this is the max page number
-s SEED, --seed=SEED  random number seed
-N, --notrace   do not print out a detailed trace
-c, --compute   compute answers for me
```

As usual, `-c` is used to solve a particular problem, whereas without it, the accesses are just listed (and the program does not tell you whether or not a particular access is a hit or miss).

To generate a random problem, instead of using `-a/--addresses` to pass in some page references, you can instead pass in `-n/--numaddrs` as the number of addresses the program should randomly generate, with `-s/--seed` used to specify a different random seed. For example:

```
prompt> ./paging-policy.py -s 10 -n 3
.. .
Assuming a replacement policy of FIFO, and a cache of size 3 pages,
figure out whether each of the following page references hit or miss
in the page cache.

Access: 5  Hit/Miss?  State of Memory?
Access: 4  Hit/Miss?  State of Memory?
Access: 5  Hit/Miss?  State of Memory?
```

As you can see, in this example, we specify `-n 3` which means the program should generate 3 random page references, which it does:

5, 7, and 5. The random seed is also specified (10), which is what gets us those particular numbers. After working this out yourself, have the program solve the problem for you by passing in the same arguments but with `-c` (showing just the relevant part here):

```
prompt> ./paging-policy.py -s 10 -n 3 -c
...
Solving...

Access: 5 MISS FirstIn->   [5] <-Lastin Replace:- [Hits:0 Misses:1]
Access: 4 MISS FirstIn->[5, 4] <-Lastin Replace:- [Hits:0 Misses:2]
Access: 5 HIT  FirstIn->[5, 4] <-Lastin Replace:- [Hits:1 Misses:2]
```

The default policy is FIFO, though others are available, including LRU, MRU, OPT (the optimal replacement policy, which peeks into the future to see what is best to replace), UNOPT (which is the pessimal replacement), RAND (which does random replacement), and CLOCK (which does the clock algorithm). The CLOCK algorithm also takes another argument (`-b`), which states how many bits should be kept per page; the more clock bits there are, the better the algorithm should be at determining which pages to keep in memory.

Other options include: `-C/--cachesize` which changes the size of the page cache; `-m/--maxpage` which is the largest page number that will be used if the simulator is generating references for you; and `-f/--addressfile` which lets you specify a file with addresses in them, in case you wish to get traces from a real application or otherwise use a long trace as input.

Questions

- Generate random addresses with the following arguments: `-s 0 -n 10`, `-s 1 -n 10`, and `-s 2 -n 10`. Change the policy from FIFO, to LRU, to OPT. Compute whether each access in said address traces are hits or misses.
- For a cache of size 5, generate worst-case address reference streams for each of the following policies: FIFO, LRU, and MRU (worst-case reference streams cause the most misses possible. For the worst case reference streams, how much bigger of a cache is needed to improve performance dramatically and approach OPT?
- Generate a random trace (use python or perl). How would you expect the different policies to perform on such a trace?
- Now generate a trace with some locality. How can you generate such a trace? How does LRU perform on it? How much better than RAND is LRU? How does CLOCK do? How about CLOCK with different numbers of clock bits?
- Use a program like `valgrind` to instrument a real application and generate a virtual page reference stream. For example, running `valgrind --tool=lackey --trace-mem=yes ls` will output a nearly-complete reference trace of every instruction and data reference made by the program `ls`. To make this useful for the simulator above, you'll have to first transform each virtual memory reference into a virtual page-number reference (done by masking off the offset and shifting the resulting bits downward). How big of a cache is needed for your application trace in order to satisfy a large fraction of requests? Plot a graph of its working set as the size of the cache increases.