**Chapter 22: Beyond Physical Memory: Policies**

A page fault occurs, you find a free page on the free-page list, and assign it to the faulting page.

When we have little free memory, 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.

**22.1 Cache Management**

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**, or maximize the number of **cache hits**.

**The average memory access time (AMAT)** can be computed as:

TM + (PMiss x TD)

Where TM­­ is the cost of accessing memory and TD is the cost of accessing disk and PMiss is the probability of cache miss.

**22.2 The Optimal Replacement Policy**

The intuition for optimal policy is if you have to throw out some page, why not throw out the one that is needed the furthest from now?

Assume that a program accesses the following stream of virtual pages: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1. The optimal policy will be demonstrated as follows:

Table

Description automatically generated

The first three miss are called **cold-start miss** (or **compulsory miss**). At each miss, we evict the page that will be used the furthest later. The hit rate overall is 6/11 = 54.5%.

However, this approach is not realistic because we do not know the future.

**22.3 A Simple Policy: FIFO**

For this policy, pages were simply placed in a queue when they enter the system. When replacement occurs, the page on the tail of the queue is evicted.

This approach is easy to implement. Using the above example, the behavior of FIFO is described as follows:

Table

Description automatically generated

The hit rate of FIFO is worse (36.4%). FIFO simply can’t determine the importance of blocks.

**22.4 Another Simple Policy: Random**

This approach simply picks a random page to replace under memory pressure. Again, this is easy to implement, but it doesn’t really try to be too intelligent in picking which blocks to evict. Considering the above example again, the behavior is described as follows:

Table

Description automatically generated

In the above example, it does better than FIFO. However, to determine its overall performance, we need to run it many times. We can see that sometimes, random is as good as optimal.

**22.5 Using History: LRU**

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.

One type of historical information could use frequency. if a page has been accessed many times, perhaps it should not be replaced as it clearly has some value. A 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.

Chart, histogram

Description automatically generated

This family of policies is based on what people refer to as the **principle of locality**, which basically is just an observation about programs and their behavior.

The **Least-Frequently-Used (LFU)** policy replaces the least-frequently used page when an eviction must take place. Similarly, the **Least-Recently Used (LRU)** policy replaces the least-recently-used page.

Table

Description automatically generated

We can see that using the above example, LRU does as well as the optimal policy.

We should also note that the opposites of these algorithms exist: Most Frequently-Used (MFU) and Most-Recently-Used (MRU). However, they do not work well.

**22.6 Workload Examples**

Here, we’ll examine more complex workloads instead of small traces. Our first workload has no locality, which means that each reference is to a random page within the set of accessed pages. The result of the experiment for the above policies and optimal is described as follows.

When there is no locality, all policies perform the same. In addition, when the cache is large enough, all policies converge to 100% hit rate. The optimal performs noticeable better than the realistic policies.

Chart

Description automatically generated

The next workload we examine is called the “80-20” workload, which exhibits locality: 80% of the references are made to 20% of the pages (the “hot” pages); the remaining 20% of the references are made to the remaining 80% of the pages (the “cold” pages). In our workload, there are a total 100 unique pages again.

Chart

Description automatically generated

LRU does slightly better now as hot pages have been referred to frequently in the past, they are likely to be referred to again in the future. However, it is still not the optimal.

LRU performance also depends on the cost of miss. If it is costly, then the difference is significant.

In out final example, the looping sequence, 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.

Chart, line chart

Description automatically generated

This represents the work case for both FIFO and LRU. Random policy does not have weird corner-case behaviors.

**22.7 Implementing Historical Algorithms**

Implementing these approaches can be troublesome. To implement LRU perfectly, we need to do a lot of work. Specifically, upon each page access, we must update some data structure to move this page to the front of the list. In addition, 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 (we can improve this using hardware support).

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.

**22.8 Approximating LRU**

Approximating LRU requires some hardware 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.

The OS can use bit to approximate LRU using clock algorithm. 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 use bit for P is set to 0 (cleared), and the clock hand is incremented to the next page (P + 1). The algorithm continues until it finds a use bit that is set to 0, implying this page has not been recently used.

**22.9 Considering Dirty Pages**

One small modification to the clock algorithm that is commonly made is the additional consideration of whether a page has been modified or not while in memory.

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. To support this behavior, the hardware should include a **modified bit** or **dirty bit**.

**22.10 Other VM Policies**

The OS also has to decide when to bring a page into memory. This policy, sometimes called the **page selection** policy.

For most pages, the OS simply uses **demand paging**, which means the OS brings the page into memory when it is accessed, “on demand” as it were.

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 ones.

**22.11 Thrashing**

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? **Thrashing**

Some earlier operating systems had a fairly sophisticated set of mechanisms to both detect and cope with thrashing when it took place. A system could decide not to run a subset of processes, with the hope that the reduced set of processes’ **working sets** 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 poorly.

Some current systems take more a draconian approach to memory overload. For example, some version of Linux runs an **out-of-memory killer** when memory is oversubscribed, which chooses a memory-intensive process and kills it. This can cause trouble if we accidentally kill the server.