# 【OS】Day28

| | |
|---|---|
| Class | Operating System: Three Easy Pieces |
| Date | @January 27, 2022 |

## 【Ch22】Swapping Policies(2)

### 22.4 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.

How Random does depends entirely upon how lucky(or unlucky) Random gets in its choices.

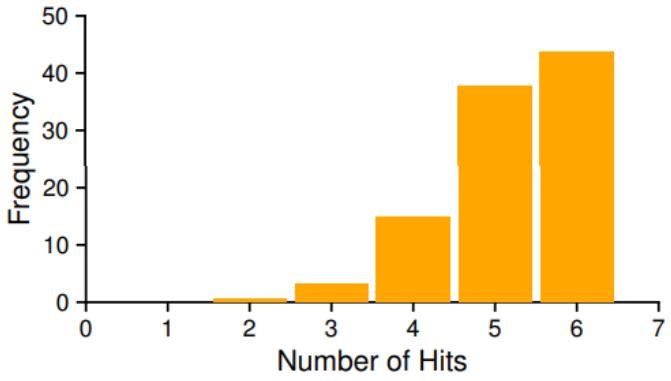


Figure 22.4: **Random Performance Over 10,000 Trials**

### 22.5 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.

As we did with scheduling policy, to improve our guess at the future, we once again learn 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 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.

- 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.

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 behaviour.

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.

The following figure shows the tracing of applying the Least-Recently-Used policy:

| 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 |

Figure 22.5: **Tracing The LRU Policy**

## 22.6 Workload Examples

The 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).

The following figure shows how the policies perform with this workload:

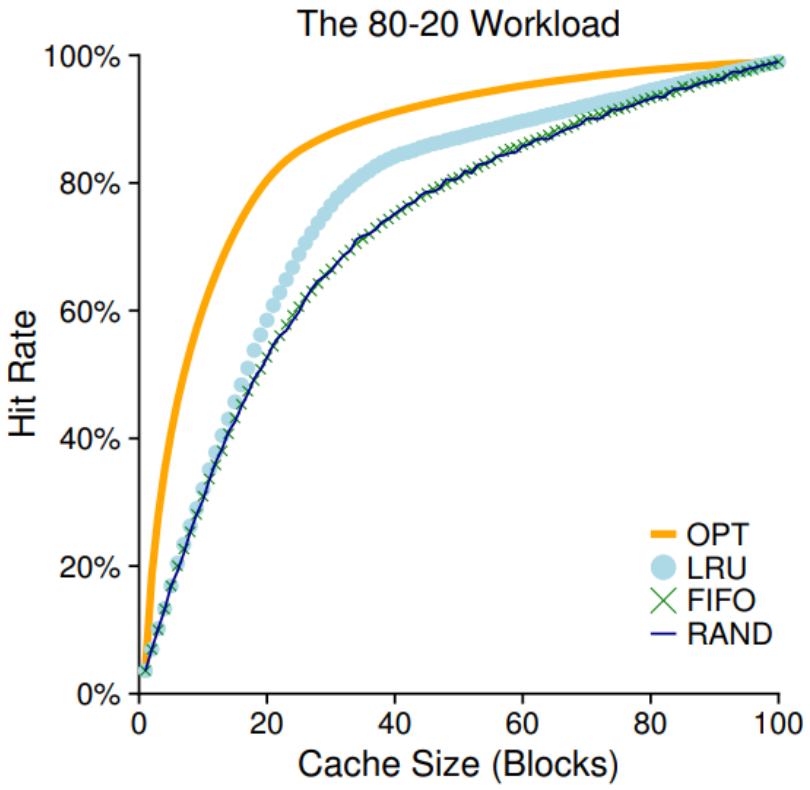


Figure 22.7: **The 80-20 Workload**

The last workload we examine is the “looping sequential” workload, 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.

This workload represents a worse-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.

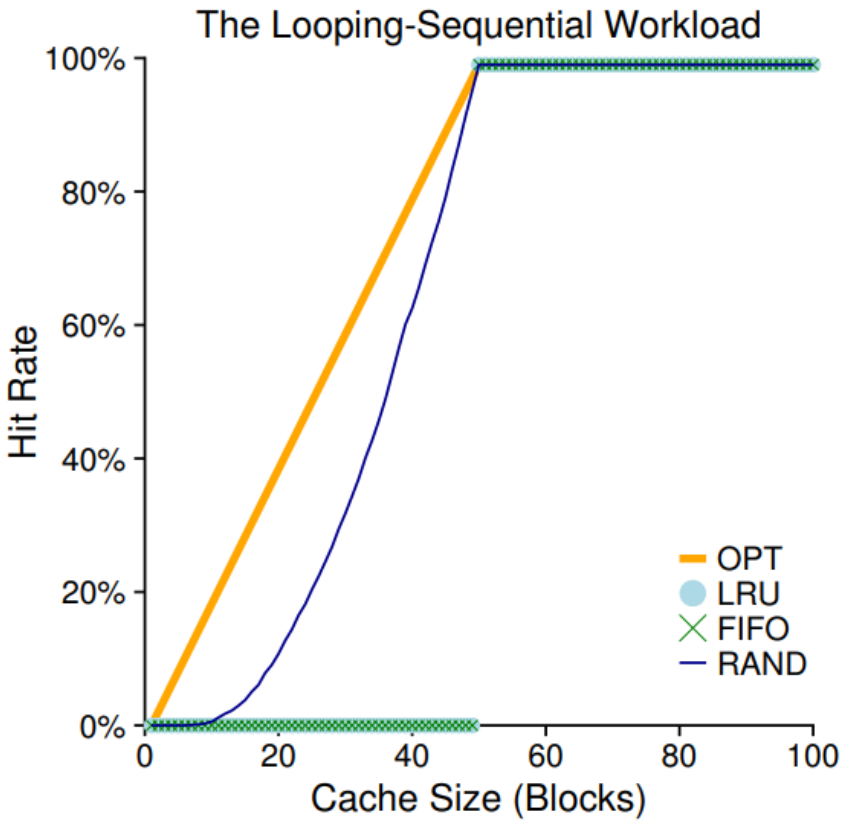


Figure 22.8: **The Looping Workload**

## 22.8 Approximating LRU

Approximating LRU is more feasible from a computational-overhead standpoint, and indeed it is what many modern systems do.

The idea 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.

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 or just in an array somewhere). Whenever a page is reference, the use bit is set by hardware to 1.

The hardware never clears the bit(i.e. sets it to 0); that is the responsibility of the OS.

*How does the OS employ the use bit to approximate LRU?*

There could be one simple approach with the clock algorithm.

A clock hand points to some particular page to begin with. 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.

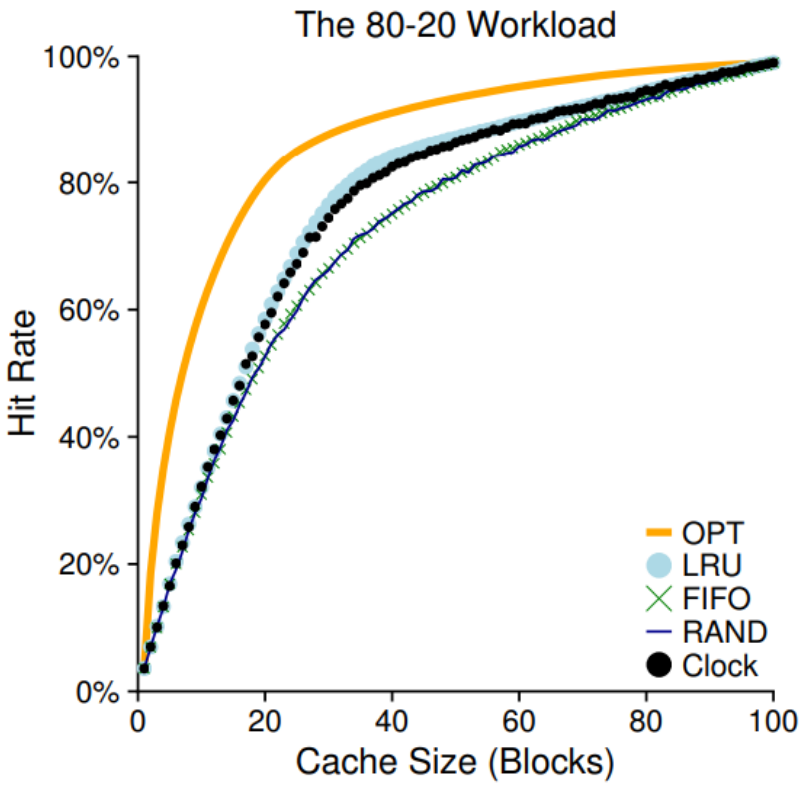


Figure 22.9: **The 80-20 Workload With Clock**

## 22.9 Considering Dirty Pages

One small modification to the clock algorithm 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.

To support this behaviour, the hardware should include a modified bit(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.

## 22.10 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, 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 demand” as it were.

Of course, the OS could guess that a pages is about to be used, and thus bring it in ahead of time; this behaviour is known as prefetching and should only be done when there is reasonable chance of success.

Another policy determines how the OS writes pages out to disk.

Of course, the 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 behaviour 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.