Virtual Memory

- 13 - Address Spaces
- 14 - Memory API
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

FIFO *another simple policy*

- Pages placed in a queue when they enter the system
- Evict page on the tail of the queue ("first-in")
  - Simple to implement, but does not care about block importance

Tracing the FIFIO 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	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

4 hits
7 misses

Reference Stream

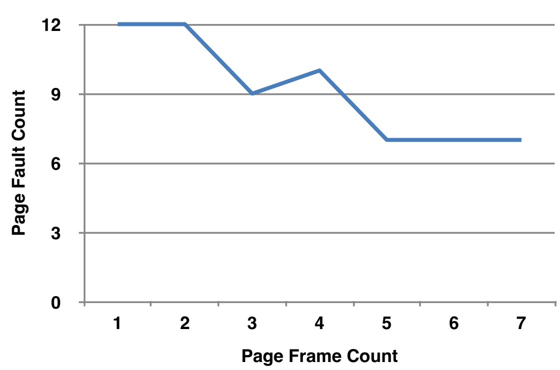
0 1 2 0 1 3 0 3 1 2 1

BELADY'S ANOMALY

- We would expect the cache hit rate to never decrease when cache grows with same input stream. But with FIFO, not so:

Reference Stream

1 2 3 4 1 2 5 1 2 3 4 5



- FIFO does not have the stack policy
  - i.e. *set of pages in n frames always subset of pages in n+1 frames*

Using History

- Lean on the past and use **history**.
  - Two type of historical information.

Historical Information	Meaning	Algorithms
recency	temporal locality says recently used page has value	LRU
frequency	Frequently used page has value, should not be replaced	LFU

Using History : LRU

- Replaces the least-recently-used page.

Reference Stream

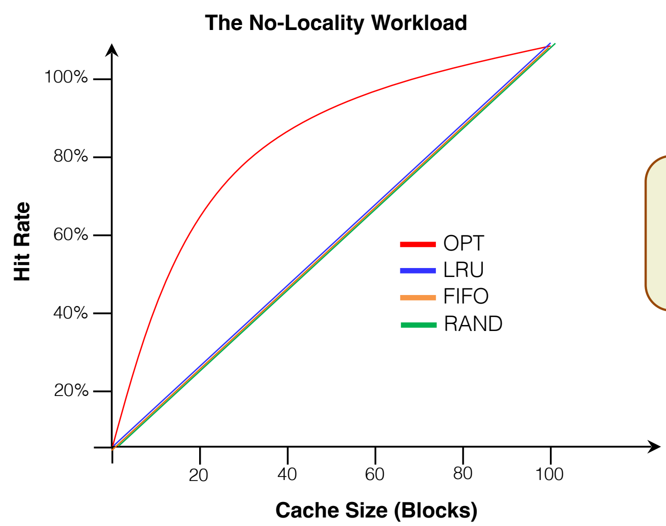
0 1 2 0 1 3 0 3 1 2 1

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

6 hits
5 misses

Workload Example : The No-Locality Workload

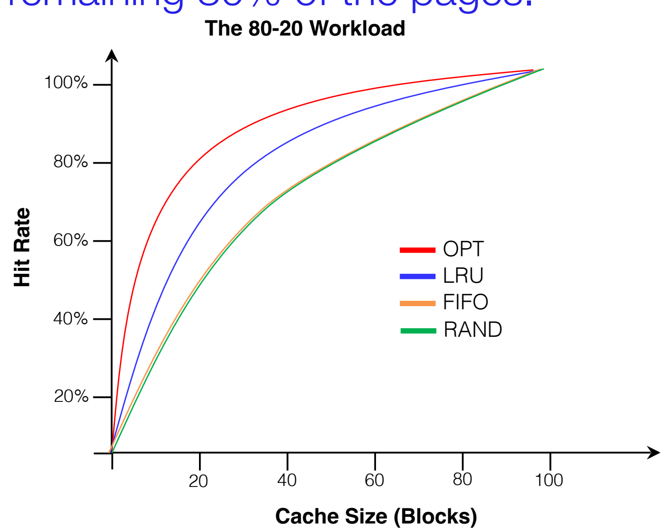
- Each reference is to a random page within the set of accessed pages.
  - Workload accesses 100 unique pages over time.
  - Choosing the next page to refer to at random



When the cache is large enough to fit the entire workload, the policy doesn't matter.

Workload Example : The 80-20 Workload

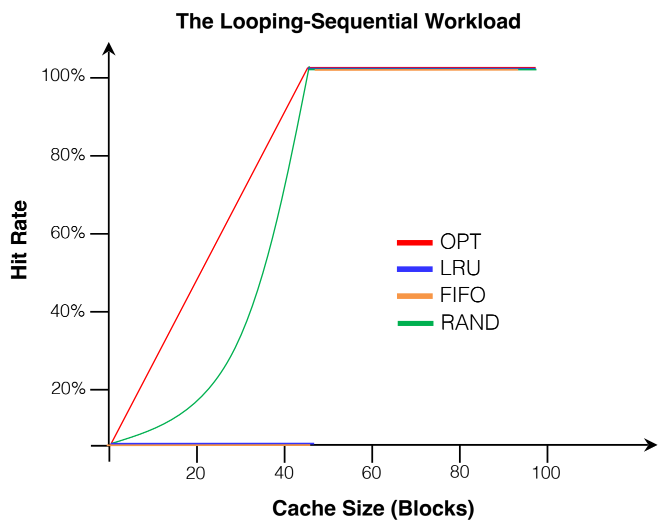
- Exhibits locality: 80% of the reference are made to 20% of the page
- The remaining 20% of the reference are made to the remaining 80% of the pages.



LRU is more likely to hold onto the hot pages.

Workload Example : The Looping Sequential

- Refer to 50 pages in sequence.
  - Starting at 0, then 1, … up to page 49, and then we Loop, repeating those accesses, for total of 10,000 accesses to 50 unique pages.



Implementing Historical Algorithms

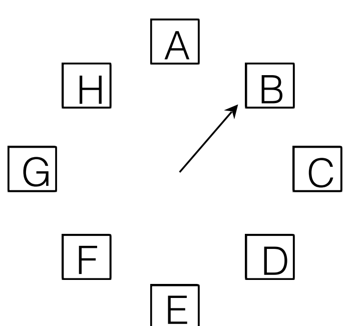
- To keep track of which pages have been least-and-recently used, the system has to do some accounting work on **every memory reference.**
  - Add a little bit of hardware support.

Approximating LRU

- Require some hardware support, in the form of a **use bit**
  - When a page is referenced, the `use` bit is set by hardware to 1
  - Hardware never clears the bit
- Clock Algorithm
  - All system pages arranged in a circular list
  - Clock hand points to the “current” page
  - When a victim is needed, pages are checked while hand is advanced:
    - if `use` = 1, `use` is set to 0
    - if `use` = 0, the page is chosen to be replaced

Clock Algorithm

- The algorithm searches for a `use` bit that is set to 0.



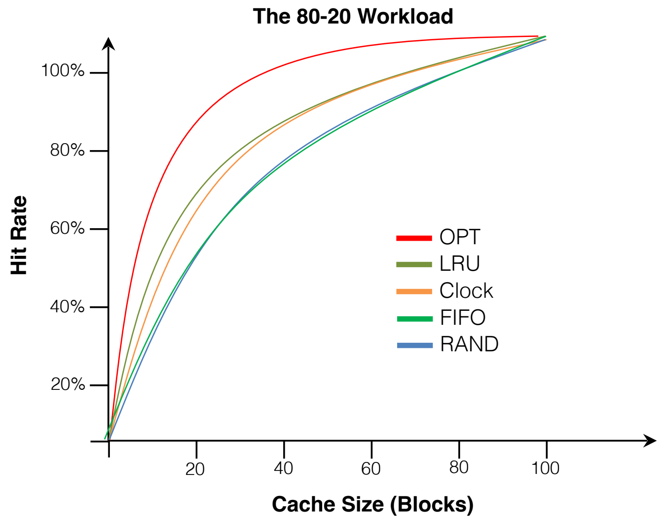
Use bit	Meaning
0	Evict the page
1	Clear `use` **bit** and advance hand

The Clock page replacement algorithm

When a page fault occurs, the page the hand is pointing to is inspected.
The action taken depends on the `use` bit

Workload with Clock Algorithm

- Clock algorithm is not a perfect approximation of LRU, but it can be close

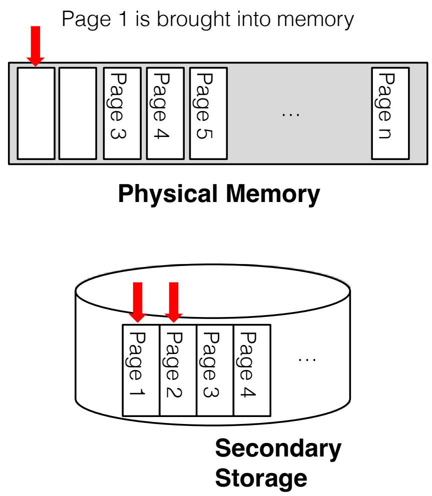


Optimizations *dirty pages*

- The hardware include a **modified bit** (a.k.a `dirty` **bit**)
  - Page has been **modified** and is thus **dirty**, it must be written back to disk to evict it.
  - Page has not been modified, the eviction is free.

Optimizations *prefetching*

- OS may fetch multiple pages from disk instead of only one



Spatial locality implies that page 2 may soon be accessed and thus should be brought into memory too

Optimizations *clustering, grouping*

- Collect a number of pending writes together in memory and write them to disk in one write.
  - A **single large write** is more efficient than **many small ones**.