Page Replacement

- the *page replacement policy* is critical for performance
  - disks are slow – page faults are expensive
  - swapping out pages that must soon be swapped back in can effectively bring the system to a halt – *thrashing*

> 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.

- can view main memory as a cache for swap space on disk
  - use terminology and metrics of cache performance to evaluate page replacement policies

Page Replacement Policies

- the optimal replacement policy is to replace the page accessed farthest in the future first
  - not implementable – future isn't known
  - provides a benchmark for evaluating other policies
    - cache hit rate = hits/(hits+misses)
    - a more realistic version is to omit *compulsory misses* – first access of any page requires bringing it into memory

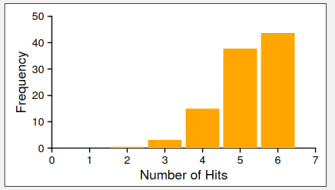
FIFO

- FIFO – replace oldest page first
  - does not take into account anything about frequency of usage
    - in fact, more frequently accessed pages are more likely to also be the oldest
    - worst case occurs when the oldest page is the next one to be used – process cycles through n+1 pages with a max of n pages in main memory
  - suffers from Belady's Anomaly
    - there are cases when a bigger cache results in a lower hit rate than a smaller cache (the opposite of the expectation)

Random

- pick a random page to evict
  - mitigates worst case by always having a 1/n chance of picking the worst page to remove
    - but also only a 1/n chance of picking the best page to remove

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



average performance over many trials is between optimal (6 hits) and FIFO (4 hits) for this sequence of memory accesses

Least-Recently-Used (LRU)

- evict the *least-recently-used* page
  - FIFO's oldest considers only the first access, ignoring more recent activity
  - LRU exploits spatial and temporal locality as a proxy for future usage
    - the most recently accessed pages are the most likely to be used 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

LRU hit rate = 6/(6+5) = 54.5%
without compulsory misses, LRU hit rate = 6/(6+1) = 85.7%

achieves same performance as the optimal policy for this sequence of accesses

Least-Frequently-Used (LFU)

- evict the *least-frequently-used* page
  - similar to LRU but does not take into account temporal locality or changing usage patterns

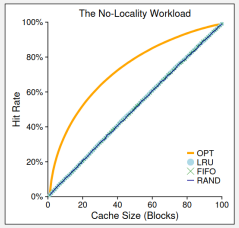
Workload Performance

- looking at a single sequence of memory accesses does not provide any real evaluation of a page replacement policy

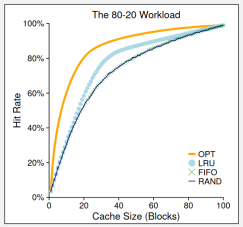
Workload Performance

10,000 page accesses
100 unique pages accessed
cache size ranges from 1-100 pages

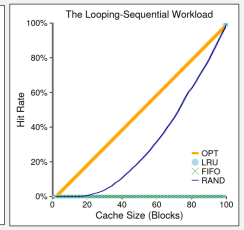
no locality – random sequence of page accesses



20% of the pages get 80% of the references



repeatedly loop through pages 0-99



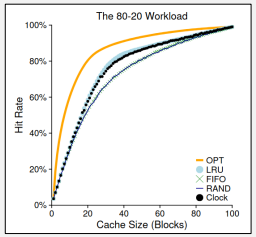
- the replacement policy doesn't matter when the cache is large enough to fit the entire workload – no misses after the first access for each page
- trying to take usage patterns into account doesn't help when usage is random
- when some pages are accessed more often, LRU is more likely to retain them
  - FIFO is based on age rather than usage – frequently-used pages are likely to also be the oldest
  - for random, a small cache means most or all of the pages it contains will be more frequently accessed ones, so there is a high chance that any page evicted will be needed again soon
- when pages are accessed in sequence, the oldest (FIFO) and the least-recently-used (LRU) pages are the same – and one that will be used again soonest
  - random mitigates the worst case

Implementing LRU

- LRU requires updating accounting info on every memory access
  - e.g. move-to-front operation in a queue of in-memory pages when a page is accessed
  - e.g. store last-access time stamp in PTE, then scan to find least recently used when eviction is needed
- even with hardware support for access time stamps, this is too expensive

Approximating LRU

- add a *use bit* (or *reference bit*) to the PTE
  - hardware sets to 1 when a page is referenced (read from or written to)
- finding a page to replace – *clock algorithm*
  - maintain a circular list of the pages in the system, with a current *clock hand* pointer
  - when replacement is required, repeatedly check the use bit of the current page
    - if 1, page has been used recently – clear the use bit to 0 and advance the clock hand
    - if 0, page has not been used recently (or all pages have been used recently, so any is equally good – or bad – to replace) – replace that page



https://pages.cs.wisc.edu/~remzi/OSTEP/vm-beyondphys-policy.pdf

Further Improvements

- if a page has not been modified while it is in memory, eviction is free
  - no need to write page back out to disk
- add a *dirty bit* (or *modified bit*) to PTE
  - hardware sets the bit when a page is written
- modify clock algorithm to also prefer clean over dirty
  - first look for unused and clock, then unused and dirty, etc

Other VM Policies

- *page replacement* – which pages to evict
- *page selection* – when to load pages from disk
  - *demand paging* loads pages when they are accessed
  - *prefetching* loads pages before they are needed
    - e.g. when loading page *P*, also load page *P*+1
    - saves overhead of invoking page fault handler twice and the delay of waiting for two separate loads
    - since page faults are expensive, prefetching should be used carefully
- when to write pages out to disk
  - *clustering* or *grouping* writes can be more efficient than doing writes individually

Other VM Policies

- controlling thrashing
  - *admission control* – system controls which processes are allowed to run, to reduce the number of pages required in memory in order to make progress
  - *out-of-memory daemon* – terminate memory intensive processes
    - drawbacks: if that process is an important one, such as your window manager…