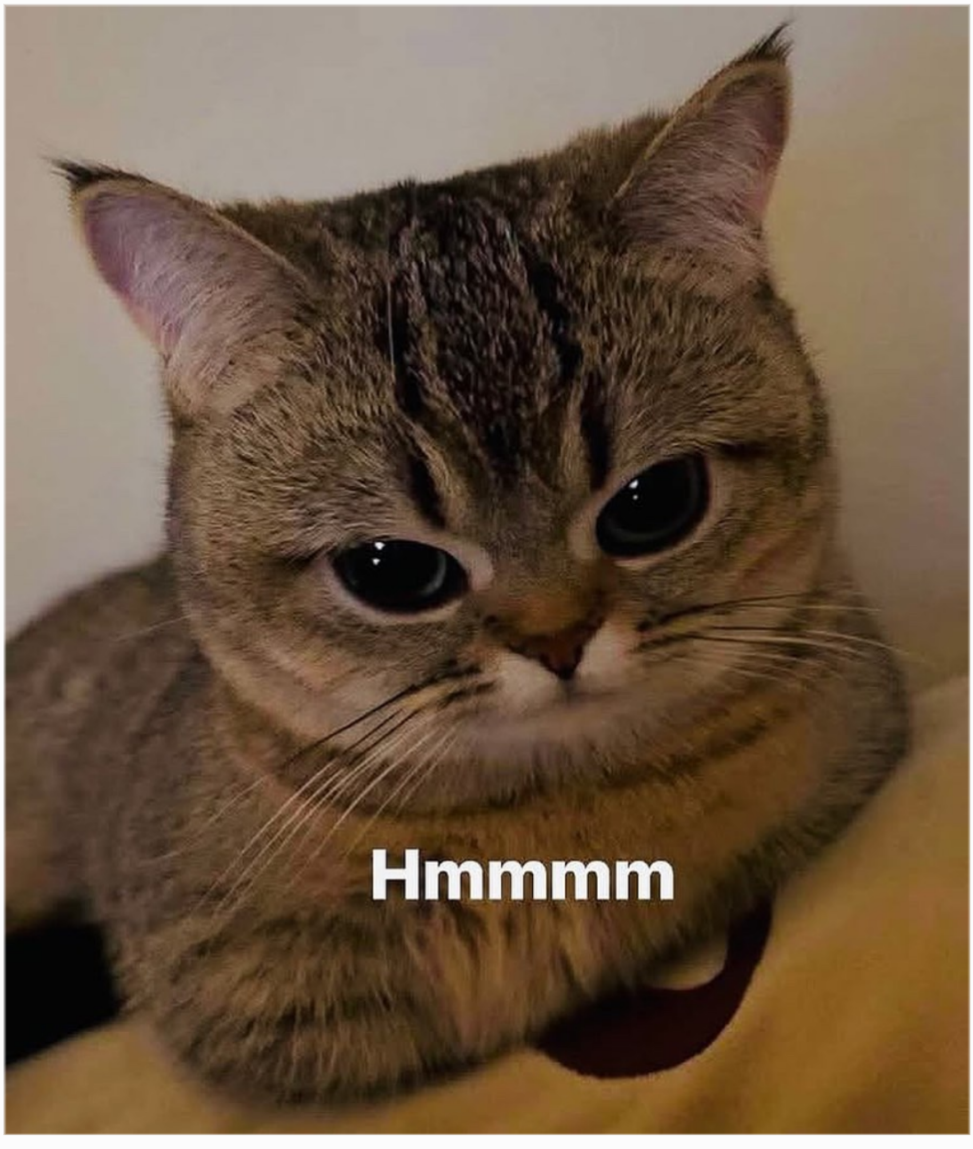
18. Swapping : Policies



when little memory is free
-> memory pressure forces the OS to start paging out pages to make room for actively-used pages

HOW TO DECIDE WHICH PAGE TO EVICT = replacement policy

main memory holds some subset of all the pages in the system
=> main memory == a cache for virtual memory pages in the system

goal of picking a replacement policy

to minimize the number of cache misses
as maximizing the number of cache hits

calculate Average Memory Access Time

$$AMAT = T_M + (P_{Miss} \cdot T_D)$$

- you always pay the cost of accessing the data in memory
- when you miss, you must **additionally** pay the cost of fetching the data from disk

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.

hit rate / we can also compute the hit rate modulo compulsory misses (= ignore the first miss to a given page)

The Optimal Replacement Policy

optimal policy : leads to the fewest number of misses overall

simple approach that replaces the page that will be **accessed furthest in the future**
/ unfortunately, impossible to implement! = future isn't known!

but it is incredibly useful as a comparison point in simulation or other studies. (measuring how it close to optimal / perfocet)

in any study you perform, knowing what the optimal is lets you perform a better comparison, showing

- how much improvement is still possible
- when you can stop making your policy better, because it is close enought to the ideal

Types of Cache Misses

1. Cold-start miss (= compulsory miss)

: the very first reference to the item. cache is not full but it doesn't have that item.

2. Capacity miss

: cache ran out of space & had to evict an item to bring a new item into the cache

3. Conflict miss

: because of limits on where an item can be placed in a hardware cache, due to set-associativity.
it does not arise in the OS page cache because such caches are always fully-associative = there are no restrictions on where in memory a page can be placed.

FIFO / avoid the complexity.
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

Another Simple Policy : Random

- simple to implement.
- doesn't really try to be too intelligent in picking which blocks to evict
- depends entirely upon how lucky
- no corner case behaviors

Belady's Anomaly

you would expect the cache hit rate to increase when the cache gets larger. But with FIFO, it gets worse!

```
replacement policy : FIFO
memory reference stream : 1 2 3 4 1 2 5 1 2 3 4 5
```

```
cache size : 3 / total cache miss : 9
[x x x] => 1 : compulsory miss
[1 x x] => 2 : compulsory miss
[1 2 x] => 3 : compulsory miss
[1 2 3] => 4 : capacity miss
[2 3 4] => 1 : capacity miss
[3 4 1] => 2 : capacity miss
[4 1 2] => 5 : capacity miss
[1 2 5] => 1 : cache hit !
[1 2 5] => 2 : cache hit !
[1 2 5] => 3 : capacity miss
[2 5 3] => 4 : capacity miss
[5 3 4] => 5 : cache hit !
```

```
cache size : 4 / total cache miss : 10
[x x x x] => 1 : compulsory miss
[1 x x x] => 2 : compulsory miss
[1 2 x x] => 3 : compulsory miss
[1 2 3 x] => 4 : compulsory miss
[1 2 3 4] => 1 : cache hit!
[1 2 3 4] => 2 : cache hit!
[1 2 3 4] => 5 : capacity miss
[2 3 4 5] => 1 : capacity miss
[3 4 5 1] => 2 : capacity miss
[4 5 1 2] => 3 : capacity miss
[5 1 2 3] => 4 : capacity miss
[1 2 3 4] => 5 : capacity miss
```

some other policies, such as LRU, don't suffer from this problem because, LRU has stack property (cache size : 3 is subset of cache size : 4)

```
replacement policy : LRU
memory reference stream : 1 2 3 4 1 2 5 1 2 3 4 5
```

```
cache size : 3 / total cache miss : 10
[x x x] => 1 : compulsory miss
[1 x x] => 2 : compulsory miss
[1 2 x] => 3 : compulsory miss
[1 2 3] => 4 : capacity miss
[2 3 4] => 1 : capacity miss
[3 4 1] => 2 : capacity miss
[4 1 2] => 5 : capacity miss
[1 2 5] => 1 : cache hit !
[2 5 1] => 2 : cache hit !
[5 1 2] => 3 : capacity miss
[1 2 3] => 4 : capacity miss
[2 3 4] => 5 : capacity miss
```

```
cache size : 4 / total cache miss : 8
[x x x x] => 1 : compulsory miss
[1 x x x] => 2 : compulsory miss
[1 2 x x] => 3 : compulsory miss
[1 2 3 x] => 4 : compulsory miss
[1 2 3 4] => 1 : cache hit!
[2 3 4 1] => 2 : cache hit!
[3 4 1 2] => 5 : capacity miss
[4 1 2 5] => 1 : cache hit!
[4 2 5 1] => 2 : cache hit!
[4 5 1 2] => 3 : capacity miss
[5 1 2 3] => 4 : capacity miss
[1 2 3 4] => 5 : capacity miss
```

algorithms with stack property, a cache of size N+1 naturally includes the contents of a cache of size N
=> when increasing the cache size,
hit rate will either stay the same or improve

FIFO & Random clearly do not obey the stack property => susceptible to anomalous behavior

Using History : LRU

any policy as simple ase 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.
=> not likely approach optimal => something smarter is needed.

=> to improve our guess at the future, we lean on the past and use history as our guide

Policies based on principle of locality

- frequency : if a page has been accessed many times, perhaps it should not be replaced as it clearly has some value / LFU
- recency : the more recently a page has been accessed, perhaps the more likely it will be accessed again. / LRU

programs tend to access certain code sequences (ex. in a loop) and data structures (ex. an array accessed by the loop) quite frequently.
=> we should try to use history to figure out which pages are important & keep those pages in memory when it comes to eviction time

locality is a good thing to keep in mind while designing caches of any kind.
but it doesn't guarantee success.
Rather, it is a heuristic that often proves useful in the design of computer systems.

Most frequently-Used (MFU) and Most-Recently-Used(MRU) : also exist in most cases (not all), these policies do not work well.
As they ignore the locality most programs exhibit instead of embracing it.

Workload Examples

if the cache is large enough to fit the entire workload, it doesn't matter which policy you use
: all policies (even random) converge to a 100% hit rate when all the referenced blocks fit in cache.

optimal performs noticeably bettern than the realistic policies
: peeking into the future, if it were possible, does a much better job of replacement

When workload has No locality,
doesn't matter much which realistic policy you are using
: LRU, FIFO, and Random all perform the same
hit rate exactly determined by the size of the cache.

When workload has locality,
LRU does better than random / FIFO.
as it is more likely to hold onto the hot pages. (use historical information)

When looping sequential workload,

- represents a worst case for both LRU and FIFO
- looping sequential workload common in many applications (including important commercial applications such as databases)
- these algorithms 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.
- Random fares notably better, not quite approaching optimal, but at least achieving a non-zero hit rate.

what if we use MFU or MRU?

- MFU : Could be better than LRU/FIFO, but not optimal
- MRU :performs badly in looping workloads.

Implementing Historical Algorithms

an algorithm such as LRU can generally do a better job than simpler policies like FIFO or random, which may throw out important pages.
: How do we implement them?

in 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)

in LRU
upon each page access, we must update some data structure to move this page to the front of the list(=the MRU 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 to speed up : add hardware support.
ex. use a time field in memory
1. a machine could update, on each page access, a time field in memory
    - in the per-process page table
    - just in some separate array in memory, with one entry per physical page of the system
2. thus, when a page is accessed, the time field would be set, by hardware, to the current time.
3. Then, when replacing a page, the OS could simply scan all the time fields in the system to find the least-recently-used page.
=> **Doesn't this scanning takes a lot of time? : YES**

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.
=> Do we really need to find the absolute oldest page to replace?
Can we instead survive with an approximation?

Given that it will be expensive to implement perfect LRU,
can we approximate it in some way, and still obtain the desired behavior?

Approximating LRU

more feasible from a computational-overhead standpoint

Use bit (= reference bit)

- requires some hardware support
- one use bit per page of the system
- 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 (r/w) the use bit is set by hardware to 1
- the hardware never clears (set to 0) the bit, though : that is the responsibility of the OS

Clock algorithm

1. a clock hand points to some particular page to begin with
2. 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
   => page P was recently used & not a good candidate for replacement
   => the use bit for P is set to 0 (cleared)
   => the clock hand is incremented to the next page (P+1)
3. the algorithm continues until it finds a use bit that is set to 0
   => 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.
- any approach which periodically clears the use bits and then differentiates between which pages have use bits of 1 vs. 0 to decide which to replace would be fine.

- The clock algorithm had the nice property of not repeatedly scanning through all of memory looking for an unused page
- doesn't do quite as well as perfect LRU, it does better than approaches that don't consider history at all.

Considering Dirty Pages

- one small modificiation to the clock algorithm
- additional considertation of whether a page has been modified or not while in memory

- if a page has been modified = dirty => it must be written back to disk to evict it = expensive
- if it has not been modified = clean => 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 overe dirty pages.

hardware should include a modified bit (= dirty bit)
this bit is set any time a page is written
=> can be incorporated into the page-replacement algorithm
ex. the clock algorithm could be changed to scan for pages that are both unused & clean to evict first.
failing to find those, then for unused pages that are dirty, and so forth

Other VM Policies
page replacement is not the only policy the VM subsystem employs (though it may be the most important)

Page selection policy
the OS also has to decide when to bring a page into memory, 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.

prefetching
the OS could guess that a page is about to be used => bring it in ahead of time

- should only be done when there is reasonable chance of success

ex. some systems will assume that if a code page P is brought into memory, that code page P+1 will likely soon be accessed
=> should be brought into memory too.

Clustering (=grouping) of writes
Determines how the OS writes pages out to disk
: they could simply be written out one at a time => expensive
=> many systems collect a number of pending writes together in memory
=> write them to disk in one more efficient write.
nature of disk drives, a single large write is more efficient than many small ones

Thrashing

what should the OS do when memory is simply oversubscribed & the memory demands of the set of runnning processes simply exceeds the available physical memory?
=> the system will constanly be paging
*also means **moving individual memory pages** (usually 4KB each) **between RAM and disk**.

admission control

- 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 => thus can make progress
- it is sometimes better to do less work well than to try to do everything at once poorly

out-of-memory killer

some systems take more a draconian approach to memory overload.
ex. some version of Linux run an out-of=memory killer when memory is oversubscribed
this daemon chooses a memory intensive process & kills it => reducing memory in a none-too-subtle manner.

- while successful at reducing memory pressure, this approach can have problems, if, for example, it kills the X server and thus renders any applications requiring the display unusable.

For many years, the importance of replacemnet algorithms had decreased, as the discrepancy between memory-access and disk-access times was so large.
because paging to disk was so expensive, the cost of frequent paging was prohibitive.
simply put, no matter how good your replacemnet algorithm was, if you were performing frequent replacements, your system became unbearably slow.
=> the best solution was a simple (if intellectually unsatisfying) one : buy more memory.

Q. but isn't choosing good replacement algorithm reduce page fault-> disk i/o dramatically?

Yes, a **good replacement algorithm helps** reduce disk I/O, but:

- A **good algorithm helps a lot when** your system has **enough memory** to keep the "right" pages.
- But when **you don't have enough RAM**, **no algorithm can save you completely** — because *something useful always gets kicked out.*

However, recent innovations in much faster storage devices have changed these performance ratios yet again, leading to a renaissance in page replacement algorithms.

even in a trace with some locality, it's still possible that LRU works worse than Random