5. buffer cache?

buffer cache: caches I/O data into RAM, often from slow storage devices.

page cache:
- pages of information brought from disk
- pages of executables/binaries loaded from disk
- pages of running processes' segments: HEAP, STACK, etc.

modern OSs, marge the buffer and page caches.

cache management:
- useful to store in faster memory, data from much slower memory
- useful if you ca justify the cost/processing of caching: if you 'hit' the cached data enough times (a random workload is not a good candidate).
- overhead, speed

- cache: a "subset copy of data from source X into location Y", where speed(Y) >> speed(X)

Issues with caches:
1. if you make a "copy", how to keep 2+ copies in sync
- is it possible that data in X and Y is different? which is newer?
- Q: how to detect a difference? timestamps?
- Q: do you detect a change all the time, or just periodically, trading off reliability (cache staleness) for performance?
- if X is newer than Y: source has changed somehow, my Y value is stale. One solution is to throw away Y, and reload it from X as needed.
- if Y is newer than X: cached data is "dirty" and needs to be flushed to its source (X). Once update to Y completes, can mark as "uptodate".

- what if both X and Y were modified? often hard to merge multiple changes, and need to involve a person to resolve the merge conflict.

2. how long should cached data remain in cache?

recall: ideal cache would have a high hit ratio. A miss means having to bring something from slow source and store it into the cache (assuming you'll get future hits on it).

But what if there's no space in cache now? Need to clear up some space in the cache and keep some reserve space -- so that new items to cache could be easily inserted into the cache (w/o having to search the whole cache to decide what to throw away).

2a. up-to-date data. may not want to keep uptodate data in cache for too long.

2b. dirty data: must flush changes to slower media, takes time (even async). Because this is slower, important to keep some reserve space for new cached items.

If a system crashes or loses power, all dirty data in cache is lost; changes to persistent media are lost. Alternative: don't keep dirty data, flush to media always, but that's slow. (Databases and f/s journals write logs synchronously, for reliability -- to replay the log/journal upon recovery.)

Flushing dirty data: if too infrequent, higher chances of data loss (reliability). If too frequent, hurts performance. How long? It depends on the trade-off b/t reliability and performance. Typically, OSs flush dirty caches every 30 or 5 seconds.

Above is about cache replacement (or reclamation) policies.

Often using Least Recently Used (LRU).
- useful to know what's been accessed recently, b/c likely to be accessed soon.
- naive way: use a timestamp for every cache access
- in OS, not using timestamps for cache mgmt

1. need a clock source w/ enough resolution. Need something like TSC register (CPU clock speed ticks). But, sampling TSC takes dozens of CPU cycles.
2. sample clock, copy to register, copy to RAM, store in some d-s alongside cache.
3. next, will have to sort all timestamps to find oldest ones. O(n*logn).

* What does the ``two hand-clock'' refer to?

LRU-like page reclamation method, that uses h/w speeds.

1. At time T1: mark all items in cache as "unused" (set a bit to 0). Can be done in OS software.

2. As the MMU starts accessing memory pages, it turns the per-page bit to 1. Done by h/w, so fast!

3. At time T2 (2nd hand of the clock): check which cached items' "unused" bit is 0 or 1. Can be done in OS software.

If an item's bit is 0: item was not used b/t T2-T1

If an item's bit is 1: item was indeed used b/t T2-T1

To reclaim: remove some or all of items with bit set to 0, b/c those are less recently used than the items with a '1'.

If T2-T1 is too small: will discard too many (possibly) all pages, and then have to re-cache some of them.

If T2-T1 is too large: may not clean enough items, not have enough free space in cache, and have to repeat two-hand clock sweep.

* What is a ``context switch''?

when cpu/scheduler wants to run a new process, has to preserve the full state of the currently running process (CPU registers and all), preserved in DRAM, and then load the new process (or an older one) with all of its preserved state.

What's a mode switch: switching b/t kernel (privileged) and user (unprivileged) mode.

* What is a ``system call''?

Code running in the kernel (in kernel mode) on behalf of a user process.
Mechanism: a trap.

* What does the unlink(2) system call?

"delete" a link to a file. Just deletes one reference to the name of the file. Doesn't delete actual data.

* What does the "ln -s" command do?

create a symbolic link.