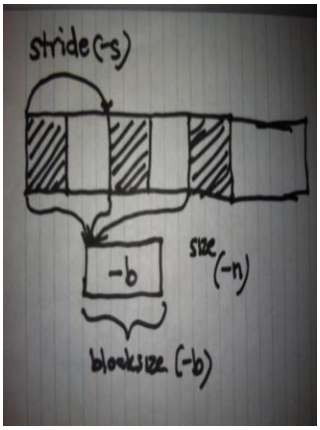
Scribe Notes, 11/8/12

Processor caches
Optimization



Looking at memory-mapped I/O

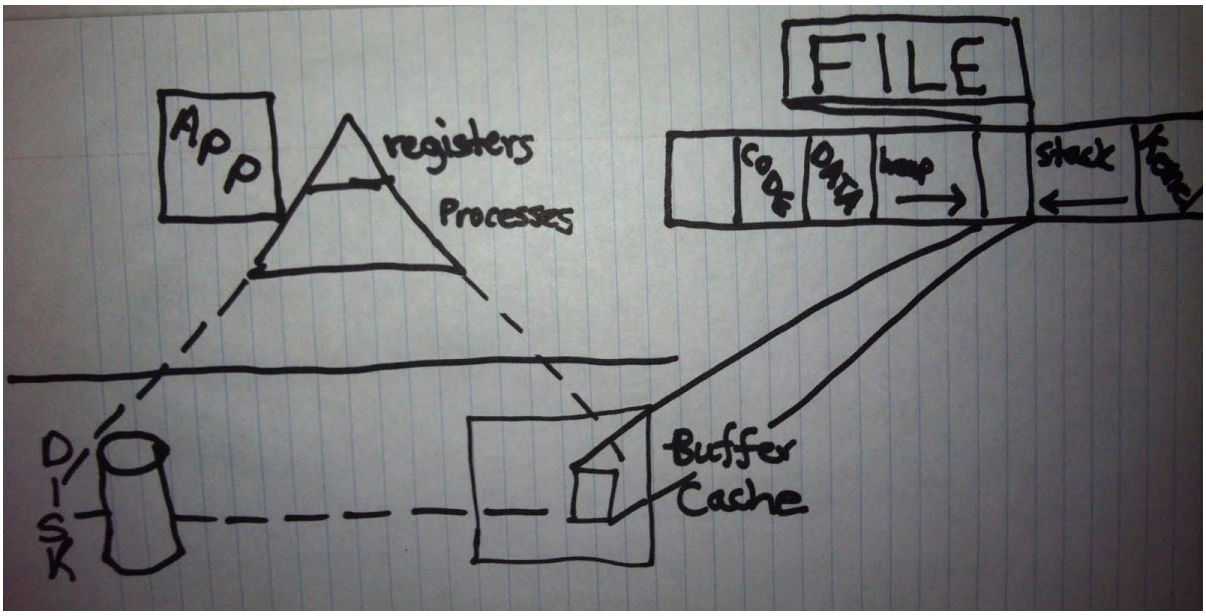
- -n argument: SIZE
- -b argument: BLOCK SIZE
- -s argument: STRIDE

Process:

1. read BLOCKSIZE of memory into buffer
2. Skip ahead by STRIDE
3. When end of memory is reached, seek back to 0 + BLOCKSIZE*i and start again

See examples in table below..

flags	what it's doing	time
-n $((100 << 20)) -b 1 -s 0	reading one byte at a time sequentially	(first run) 7.8 * 10^7 B/s (next time) 1.7 * 10^8 B/s
-n $((100 << 20)) -b 2 -s 2		faster than above..only because it was already in the computer's memory cache from running it the first time



What happened when we changed stride size to two?

- Buffer cache is used for every run of the application
- mmap acts as a call to malloc, allocating free space in the buffer cache
- Kernel (which owns the buffer cache) must wait until the entire file can be read into the buffer cache
- Well written kernel will never make redundant runs into the buffer cache
- During the first run of the program, the buffer cache was *cold*

COLD

- On the 1st run of a program, buffer cache is cold
- Cold = cache doesn't contain needed data
- When a request misses due to the cache being cold, it's known as a *cold miss*
- Note: buffer cache is fully associative

How does buffer know that we're reading the same file?

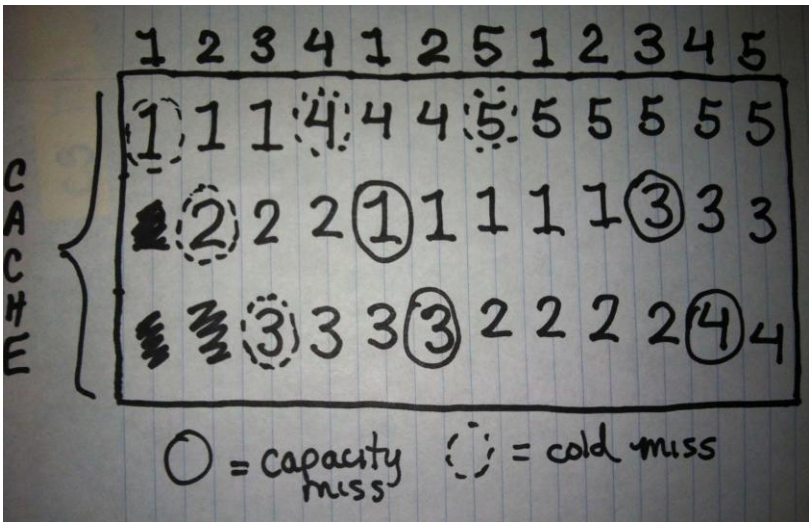
- i.e. what would justify the kernel keeping a recently read file in memory?
- A: if your program has high locality, it'll be likely to access data that it has recently accessed
- Kernel will not remove data from the cache unless it needs to use it for another purpose
    - i.e. run the program on a gigabyte file which will hopefully "evict" the data from the previous file

EVICTION

- Removing blocks from the cache to make room for other blocks
- Thus far, we have not been able to force eviction
    - We ran the program with a ten-gig file, but the ten-meg file had not been evicted from memory

EVICTION POLICIES

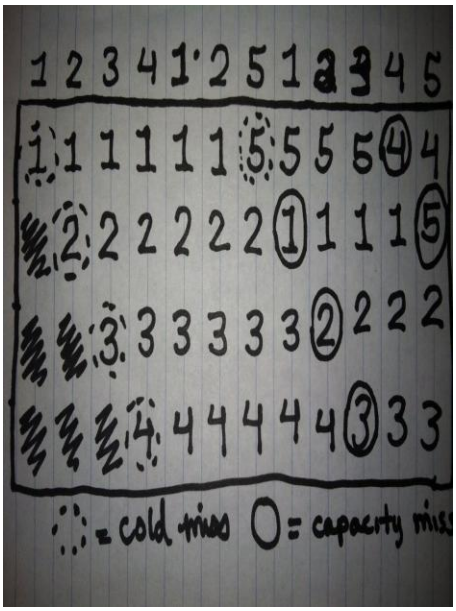
- Takes as input a reference string
    - Past accesses by block
    - Current cache contents
- Outputs a slot to evict
- <diagram of eviction policy>



    - Miss rate: 9/12 (75%)
- Optimal eviction policy
    - Evict item not needed for longest in future (not very plausible since we can't tell the future)
- FIFO policy
    - First In, First Out
    - Evict item <u>loaded</u> furthest in the past
- Cold misses
    - A miss that's required because the cache has not loaded the piece of data yet
- Capacity misses
    - A miss that's required because the cache is too small for the working set

Question: Does it take longer for the kernel to load up memory?
Answer: We cannot fix cold misses. We can try to fix capacity misses by increasing the cache size.

<diagram: same reference string, but four slots in the cache instead of three>



cold misses = 5
capacity misses = 5):
Miss rate for this diagram is 10/12 (80%) -- so increasing the cache size didn't help!
---> some eviction policies where when the cache increases, the miss rate worsens

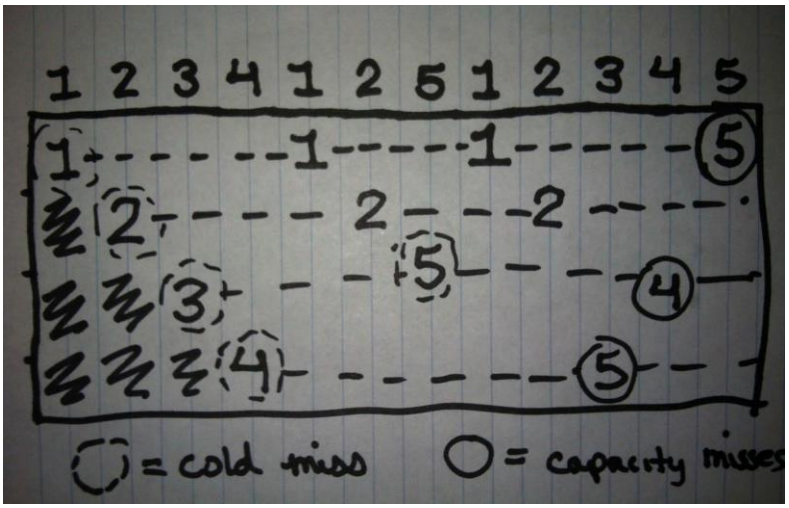
Break time

… and we're back!

LRU = Least Recently Used

- evict the item USED furthest in the past

< diagram : same reference string and cache size, but using LRU eviction policy >



cold misses = 5 (as usual)
capacity misses = 3

- This time, miss rate is only 8/12 (2/3)
- Almost every cache uses LRU as its base algorithm!
- LRU doesn't suffer from the anomaly: larger cache sizes always have the same or smaller miss rate
- Note that LRU policy is not optimized

Back to memreader program now...

flags (changing stride amt.)	what it's doing	rate (in B/s)
-n $((100 << 20)) -b 1 -s 0	reading one byte at a time sequentially	1.9 * 10^8
-n $((100 << 20)) -b 1 -s 2		1.8 * 10^8
-n $((100 << 20)) -b 1 -s 4		1.8 * 10^8
-n $((100 << 20)) -b 1 -s 64		1.5 * 10^8
-n $((100 << 20)) -b 1 -s 1024		4.7 * 10^7
-n $((100 << 20)) -b 1 -s		***1.07 * 10^8***

Reason about the performance of the reference string, and what the reference string means to the working set size of the cache.

Processor Cache

- Unit of transfer between primary memory and processor caches is the **cache line** (line = block)
    - For the purposes of this class, **cache line** has a size of 64 aligned bytes
    - how many cache lines fit in one page? 4096/64 = 2^6 = 64 cache lines / page
- Most processor caches are **SET ASSOCIATIVE**
    - Divided into groups of slots; a given cache line can fit into only one of these sets
    -

< insert diagram of sequential access (stride = 0) >

misses = 100 * 2^20 / 64 = 100 * 2^14
capacity misses = 0 (“every time we access a byte..” lol)
miss rate = 1/64

< insert diagram of stride = 2 >
cold misses (occur in the first half) = 100 * 2^20 / 64
likely # capacity misses (occur in the second half) = 100 * 2^20 / 64
total misses = 100 * 2^20 / 32 (twice as many as in stride 0 example)
miss rate = 1/32 (twice as big) -- BUT not that much slower than stride 0 example
Why?

- Processor assumes a sequential access pattern
- Stride increases ⇒ # capacity misses goes up ⇒ hit rate goes down

misses = 100 * 2^20 / 64
capacity misses = 0
miss rate = 1/64

Note: cold misses are not as bad as capacity misses
Note: today, we have in the megabytes of cache (but there is an upper bound because of cost)

The sequential cache pattern and the one we’re looking at right now are equally good on the cache

Address patterns node;
… now looking at memreader_prefetch.c

- It turns out that x86 can do prefetefching
- _ _ builtin_prefetch() - taking that address and moving it into the cache
- Trying it with our worst performing stride...
  - Without prefetching, 4.7 * 10^7
  - With prefetching, 7 * 10^7
- Every level of caching offers the ability to prefetch
  - posix fadvise = system call that says “I’m going to need this soon!”

- s 1024

0	1024	2048	…	100*2^20 - 1024
-> 1	-> 1025	-> 2049	...	-> 100*2^20 - 1023
->-> 2	->-> 1026	->-> 2050	...	->-> 100*2^20 - 1022
->->-> 3	->->-> 1027	->->-> 2051	...	->->-> 100*2^20 - 1021

how many cache lines do we store?
100* 2^20 / 2^16 = 100(2^4)