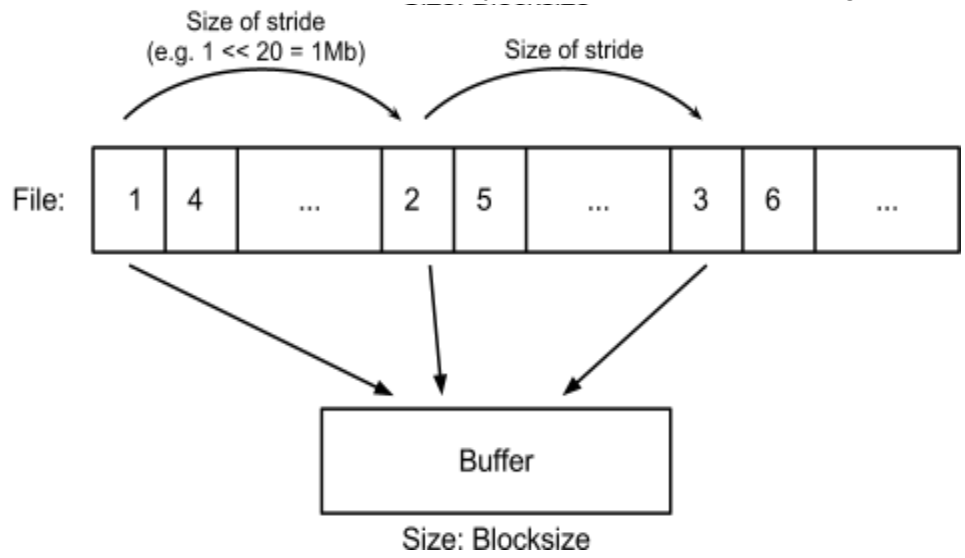
Lecture 19: Processor Caches and Program Optimization

Tracy Lu & Styliani Pantela

l18/memreader.c

This is a program that reads a large memory mapped file and it takes 3 arguments:
-n (SIZE: number of bytes that we can read)
-b (BLOCK SIZE: number of bytes that we read in every step)
-s (STRIDE)
The way the program works is that we have a file and a buffer. The size of the buffer is block_size. We read the first block of the file into the buffer. Then, we skip ahead by the stride. If the stride is 0 we read sequentially. Else, we skip, say by a MB, every time, and when we reach the end of the file, we go back to the beginning and cover the parts of the file that we haven't yet covered. We do that until we read n bytes, where n is the SIZE argument to memreader.



Examples:
–n $((100<<20)) -b 1 -s 0
These arguments would cause the file to be read sequentially in units of bytes.
Note: Memory mapped I/O is just a memory copy.

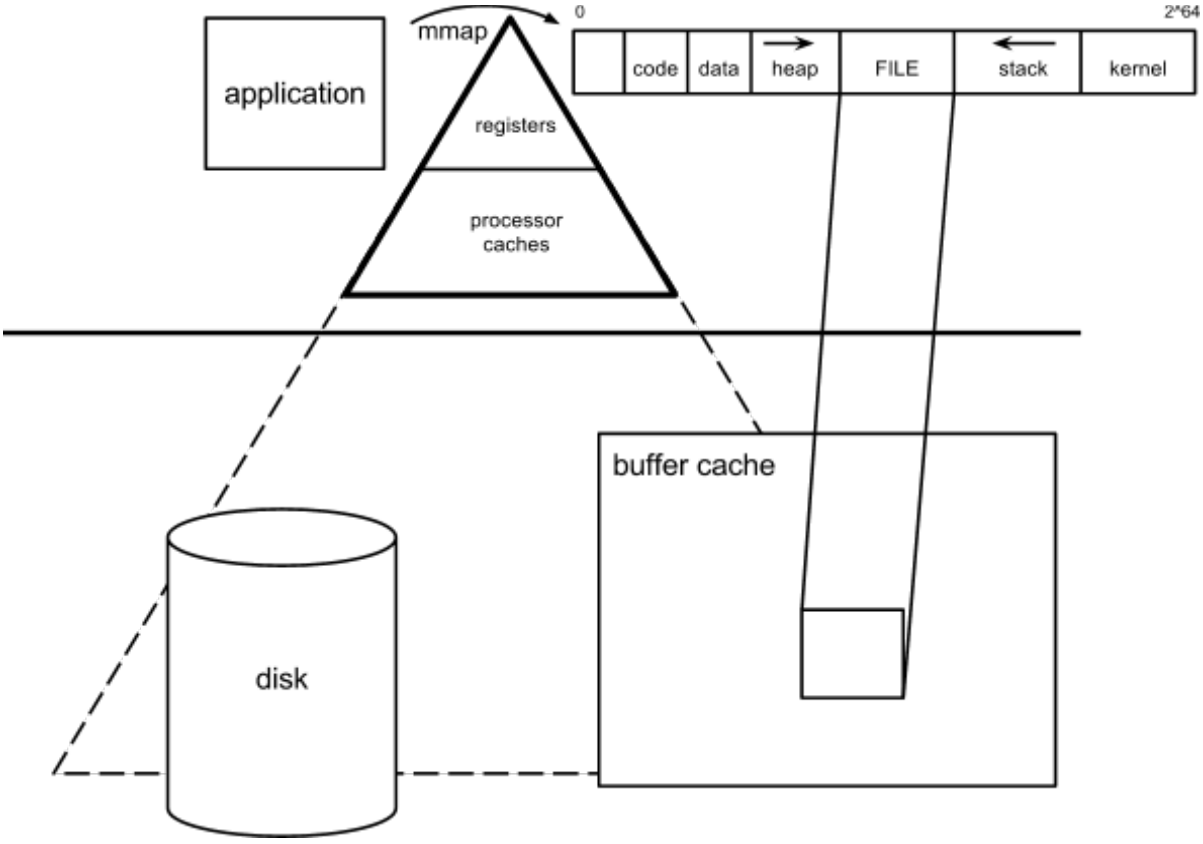
Let's actually run our memreader program now:
./memreader –n –s $((100 << 20)) –b 1 –s 0 text100meg.txt (reading on a 100mB file)
This took 7.8*10^7 Bytes/second
That was by far the largest time we've gotten. It runs a little slower from lecture 17 code because of the many arguments.

Raising the stride? Run it on stride 2. Speed goes up. But why did it go up? Let's look at the inner loop of memreader.c The amount of computation we are doing in a stride of 2 is more complex as can be seen at the end of the while loop. But it does not account for all the difference in time.

```
while (n < size) {
        size_t amount = block_size;
        if (pos + amount > size)
        amount = size - pos;

        ++nrequests;
        memcpy(buf, &file_data[pos], amount);

                n += amount;
                if (print_frequency && n - last_print >= print_frequency) {
                        report(n, tstamp() - start);
        last_print = n;
        }

        if (stride != 0) {
        pos += stride;
        if (pos >= size)
                pos = (pos - size) + block_size;
        } else
        pos += amount;
        }
```

We run the code on the sequential version again, i.e. with stride = 0. The sequential version is faster than what it was 2 minutes ago. What is happening? Is Eddie performing tricks?

Explanation: That file was not in the buffer cache the first time we run it. Remember to keep in mind all the different caches.



Every time one of this applications runs, it will have different registers. But the buffer cache is shared by all runs of the application. The first time we run the program we call mmap. Mmap is a call to malloc. Mmap allocates space in the heap that maps directly to the buffer cache. The file maps into a portion of the buffer cache. As we access parts of the file, the kernel needs to wait until the file has been read into the buffer cache.
On 1st run the buffer cache was COLD. The cache is cold when it does not currently contain the needed data. A cold miss is a cache miss that happens because the cache is cold.

Question: What part of the system understands the contents of the Buffer cache?
Answer: The kernel. The kernel owns the buffer cache and keeps it coherent relative to the disk. It will never make redundant loads into the buffer cache. If the buffer cache contains the right data,, the kernel will map it to the application space without a cold miss.

To clarify, this type of cache is fully associate.

Question: How does the kernel know that the file we want to read is the same file as the previous?
What assumption would justify the kernel keeping a recently read file in memory? If your program has high locality then it is going to be likely to access data that it just accessed again. So, it is not in the kernel's best interest to remove it from the cache unless it needs the cache for some other purpose. Either wait several hours or look at a different file, like a GB file and then

there is a chance that it will evict some parts of the 100MB file it read earlier in order to have space for the 1GB file.
After running the 1GB file, we run the 100MB file again with no stride, but we see that the 1GB file did not throw the 100MB file out of the cache since time did not change significantly. We need a bigger file.

EVICTION is removing blocks from the cache ($) to make room for other blocks. In general we evict blocks from the $ only when we need that space for other blocks. How do we choose what to evict?

EVICTION POLICIES

- Input: REFERENCE STRING which is the past by block accesses + current $ contents
- Output: Slot to evict

Experiment 1.
We draw a cache that starts out empty - COLD and has three slots. It is fully associate, that means that any block goes to any slot. Let the reference string be: 123412512345. This is an abstraction. These numbers could correspond to disk blocks, memory addresses in physical memory, web page addresses... but all these are abstracted to numbers.
Which element causes the first eviction in a 3 slot cache? The fourth different one, because we would use the first three lines for the first three elements.

Reference string:

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

Corresponding 3-slot Cache:

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

blue = cold miss
red = capacity miss

The first three reads have caused a total of 3 COLD misses. The cold misses are in blue. We need to read 4, so we have to evict something. Optimally, we would evict 3, because it will be used much later.

Optimal eviction policy evicts the item that will not be used for the longest amount of time in the future. We have a problem: optimal eviction policy REQUIRES predicting the future, which we cannot always do.

We evict 1 by the principle of locality since 1 was used the furthest into the past. Let us evict the thing that we loaded the furthest in the past. **This is the FIFO policy (First in first out). We evict the item loaded furthest in the past.**

The cache started out empty, so there are at most 5 cold misses, one for every time we introduce a new item. The others are **capacity misses.**
Capacity miss: is a miss that is required because the $ is too small for the working set. The working set is the data items the code works with. Here it is {1,2,3,4,5}.

Question: Is the reference stream passed in full to the kernel? The kernel can remember what happens in the past but does not know what happens next. Sometimes it might say 1 followed by 2? Then it is sequential access, but it might not figure out the pattern.

What is the miss rate for this reference stream?
It is 9/12 = 75%.

We cannot fix cold misses. We can fix capacity misses by getting a larger cache.

Reference string:

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

Corresponding 4-slot Cache:

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

blue = cold miss
red = capacity miss

Miss rate = 10/12 ≈ 80%. This is an ANOMALY because although we increased the slots of the cache the miss rate went up.

Break

NEW algorithm for Cache Eviction: LRU – Least Recently Used
We evict the item used the furthest in the past, NOT LOADED BUT USED. LRU is used fundamentally by every cache and it does not suffer from the anomaly we saw. This means that larger cache sizes always have the same or smaller miss rate.

Reference string:

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

Corresponding 4-slot Cache:

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

blue = cold miss
red = capacity miss
green = hit

We have five cold misses and only 3 capacity misses. Now, the miss rate is 8/12 = 66%.

Question: Is LRU optimal? NO. Optimal policy would have been to evict one or 2, when we reached 3 4 5 at the end of the reference stream. That would leave 4 and 5 in the cache.

PROCCESSOR/ BUFFER CACHE all blocks have the same size: 1page size
PROCCESSOR BLOCK SIZE: 64 bytes big

Note: When you want to find the time it takes a program to execute it is better to run it multiple times and get the median of the values we get back.

Processor cache

1) The unit of transfer between primary memory and processor caches is the cache LINE, here a line means same as block. SIZEOF(cache line) = 64 aligned bytes. Aligned meaning that the first byte has address multiple of 64. It is a workable constant and it is not as permanent a constant as the size of the byte. **In one page we have 4096/64 = 2^12/2^6 = 65 cache lines.**
2) Most processor caches are SET Associative, which is a mixture of direct and fully associative mapping. The cache is divided into slots, into groups of slots and a given block can fit into only one of those groups. Those groups are called sets.

Let's go back to the memreader.c program that reads 100MB file with different stride patterns.

Stride numbers table on 100MB file:
-n 100*2^20 –b 1 –s X where X is a number

0	1.9*10^8	B/S
2	1.8*10^8	B/S
4	1.8*10^8	B/S
64	1.5*10^8	B/S
1024	4.7*10^7	B/S (Things are falling off.)

2^20 1.03*10^8 B/S (WHY?)
After it finishes striding the file once it returns to consecutive access pattern.

Example: Stride 0
We have 100 *2^20 accesses. This has a very high locality, because we are accessing the same cache line. 64 times. The first is a COLD MISS. The following 63 are hits. The 64 is another COLD MISS. Then we have 63 more hits.

0	1	2	3	...	64	65	...
Cold Miss	Hit	Hit	Hit	Hit	Cold Miss	Hit	

Statistics
#misses = 100*2^20/64
#capacity misses = 0 - every time we access a byte we access it for the first time
#miss rate = 1/64

Example: Stride 2

1st pass through file:

0	2	4	...	64	...	(100*2^20)-2
Cold Miss	Hit	Hit	Hit	Cold Miss		Hit

2nd pass through file:

1	3	5	...	65	...	(100*2^20)-1
Cap. Miss	Hit	Hit	Hit	Cap. Miss		Hit

0 is a cold miss. Then we have hits until 64 because everything is in the same cache line. 64 is a cold miss and the same pattern repeats. In the first section we have **100*2^20/4 cold misses and the miss rate is 1/32. In the second section, we have a capacity miss on 1, again at 65, and etc.**

Statistics
#misses = 200*2^20/64
#capacity misses = 100*2^20/64
#miss rate = 1/32 (twice as big)

The processor does not give us a factor of 2 worst performance. The processor does something smart behind the scenes. Both access patterns look sequential in terms of access lines. When

it notices a sequential access pattern **it should pre-fetch**. As the stride increases #capacity misses goes up and the hit rate goes down until we heat a crossover point. A MB is past the crossover point. What is the minimum size of a processor cash that could handle a stride of 2^20 well? That is 100 lines. 100 lines = 64 hundred bytes, then it will handle this access pattern pretty well.

Example: Stride 2^20

0	2^20	2*2^20	3*2^20	...	99*2^20	1	1+2^20	1+2*2^20	...
CM	CM	CM	CM	CM	CM	H	H	H	H

2	2+2^20	...	63	63+2^20	...	64	64+2^20	64+2*2^20	...
H	H	H	H	H	H	CM	CM	CM	

CM = cold miss
H = hit

The 0 is a cold miss, everything is a miss until 99*2^20. Then everything is a hit, until 64 where we have cold misses only to have hits again!

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

This cache pattern and the sequential cache pattern are equally good on the cache on this level of abstraction. At other levels of abstraction they are not equal as we saw in our experiment above.

The x86 instruction set lets us prefetch.

./memreader-prefetch program
```
__builtin_prefetch(&file_data[pos + stride]);
```
Informs the processor that in the future this program is going to access an address and it says what that address is. Unlike an actual memory load, the processor takes that address and moves it into the cache.

```
while (n < size) {
    if (prefetch && pos + stride < size)
    __builtin_prefetch(&file_data[pos + stride]);

    size_t amount = block_size;
    if (pos + amount > size)
```

```
                amount = size - pos;

                ++nrequests;
                memcpy(buf, &file_data[pos], amount);

                n += amount;
                if (print_frequency && n - last_print >= print_frequency) {
                report(n, tstamp() - start);
                last_print = n;
                }

                if (stride != 0) {
                pos += stride;
                if (pos >= size)
                        pos = (pos - size) + block_size;
                } else
                pos += amount;
        }
        total += n;
}
```

No prefetching and 1MB file: 1.03 B/S
With prefetching and 1MB: 1.04 (performance is raised but not by match)

With stride 1024, which was our worst performing stride before, performance is improved by about a third by adding a pre-fetch instruction. By about 4.5 to roughly 7 B/S.

HOW DID THIS MAGIC HAPPEN?

-s 1024
1st pass through file:

0	1024	2048	...	100*2^20-1024

2nd pass through file:

1	1025	2-49	...	100*2^20-1023

… and then 3rd pass, 4th pass, etc.

For every pass through the file, 100*2^20/1024 cache lines are loaded, which is quite large (100kB). Thus, there's a chance that some of the processor caches will suffer capacity misses when we return to the next row. But we can pre-fetch. Before accessing one data, ask processor to pre-fetch the next data. That is, before we read number 0 we ask the processor to pre-fetch the element 1024, before we read 1024, we pre-tetch 2048, etc. This is **explicit pre-**

fetching. We often think of it as happening behind the scenes but in fact every level offers you the ability to pre-fetch explicitly and improve performance. In the processor there is a pre-fetch instruction. For the buffer cache, system calls pre-fetches.

Overview

Cold misses or compulsory misses: misses that happen because the element we are trying to read has not been loaded to cache

Capacity miss: is a miss that is required because the $ is too small for the working set. The working set are the data items the code worked with. In our examples that was: {1,2,3,4,5}.

EVICTION is removing blocks from the cache ($) to make room for other blocks. In general we evict blocks from the $ only when we need that space for other blocks. How do we choose what to evict? We choose an eviction policy.

EVICTION POLICIES

- **Optimal eviction policy evicts the item that will not be used for the longest amount of time in the future.** We have a problem: optimal eviction policy REQUIRES predicting the future, which we cannot always do.
- **FIFO policy (First in first out). We evict the item loaded furthest in the past.**
- **LRU – Least Recently Used:** We evict the item used the furthest in the past, NOT LOADED BUT USED. LRU is used fundamentally by every cache. When using LRU, the larger the cache size the same or smaller the miss rate.

Processor cache

- The unit of transfer between primary memory and processor caches is the cache LINE. SIZEOF(cache line) = 64 aligned bytes. Aligned meaning that the first byte has address multiple of 64. It is a workable constant and it is not as permanent a constant as the size of the byte. **In one page we have 4096/64 = 2^12/2^6 = 65 cache lines.**
- Most processor caches are SET Associative, which is a mixture of direct and fully associative mapping. The cache is divided into slots, into groups of slots and a given block can fit into only one of those groups. Those groups are called sets.

The x86 instruction set lets us prefetch.

Look at the ./memreader-prefetch program

```
__builtin_prefetch(&file_data[pos + stride]);
```

This instruction informs the processor that in the future this program is going to access an address and it says what that address is. Unlike an actual memory load, the processor takes that address and moves it into the cache.