note @358 stop following 3 views

Actions

CACHELAB: Tips & stuff I've learned about

Spring has sprung! Hope you all have had a nice spring break, Hope you all are enjoying the nice weather and extra sunlight. Anyway this assignment is also really cool, probably my second favorite?

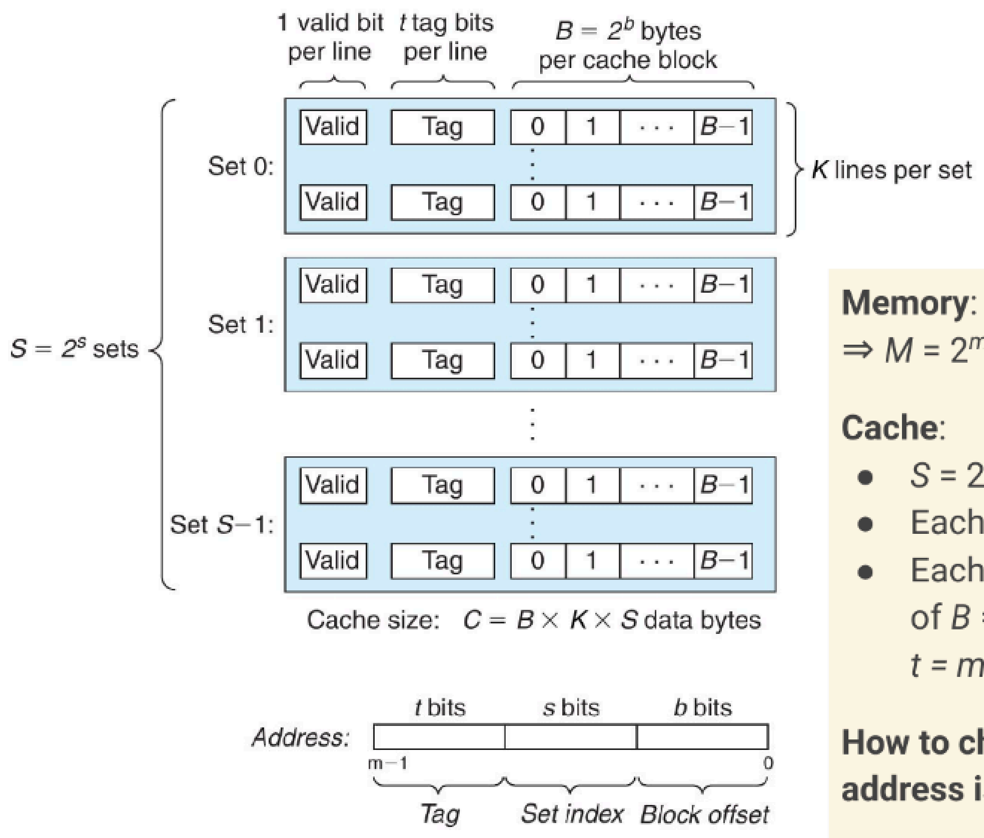
Intro

You are designing a cache and coding its protocols in C. Cache / caching is a somewhat general term, it exists in a lot of places. It is data storage, but specifically storage that we suspect to be **accessing frequently**, so we want it to be storage that we can also **access quickly, to maximize performance**. For example, if we are going to accessing / modifying the same data multiple times in a row during a program, we want to **temporarily store that data on the CPU instead of having to fetch it from the hard drive** over and over (takes much longer). That's why it's also important to be smart about what data we are storing in the cache -- in our example, CPU storage is very valuable as it has to exist on that tiny chip which also has to do a bunch of other really important stuff, while hard drive storage is cheap, that's all they do, and I can buy a 1 TB hard drive for $50 that sits in my computer and stores like 200 movies (not me personally of course). It wouldn't make sense to store parts of a movie on a CPU because it **has other things to be doing constantly and repeatedly**.

This is why it is good to **design your caching protocols considering the principles of temporal and spatial locality** (Unit 10, Slide 6) as that data is what we're likely to access frequently.

The components of the cache

Our cache is made up of **the cache itself, `S` sets in the cache, `K` lines in each set, and `B` bytes of data in each line.** However, we are just simulating the cache, not storing any actual data. Create your data structures accordingly. **Dynamically allocate everything,** otherwise you will get errors because we don't know how big our cache is until we run the program and parse the `S K B` arguments.



Memory: addresses of m bits $\Rightarrow M = 2^m$ memory locations

Cache:

- $S = 2^s$ **cache sets**
- Each set has K **lines**
- Each line has: **data block** of $B = 2^b$ bytes, **valid bit**, $t = m - (s+b)$ **tag bits**

How to check if the word at an address is in the cache?

^This is really important. Stare at it. Understand it. Everything about it.

Hint: Notice the structure of the address. The **block offset doesn't determine whether we have a hit or a miss** because all the blocks in a line of memory are a package deal -- they are all in or they are all out. Therefore, **two different addresses could be accessing the same line**.

The memory traces explained

Let's break down the contents of a memory trace.

First, we have **the instruction.** From the programming rules, we are told that we are only **focusing on the data cache performance,** which means we are **ignoring the instruction (`I`) cache calls**. We could also store instruction codes -- it might be helpful to store instructions on the cache if we are looping -- but we are ignoring that for this assignment.

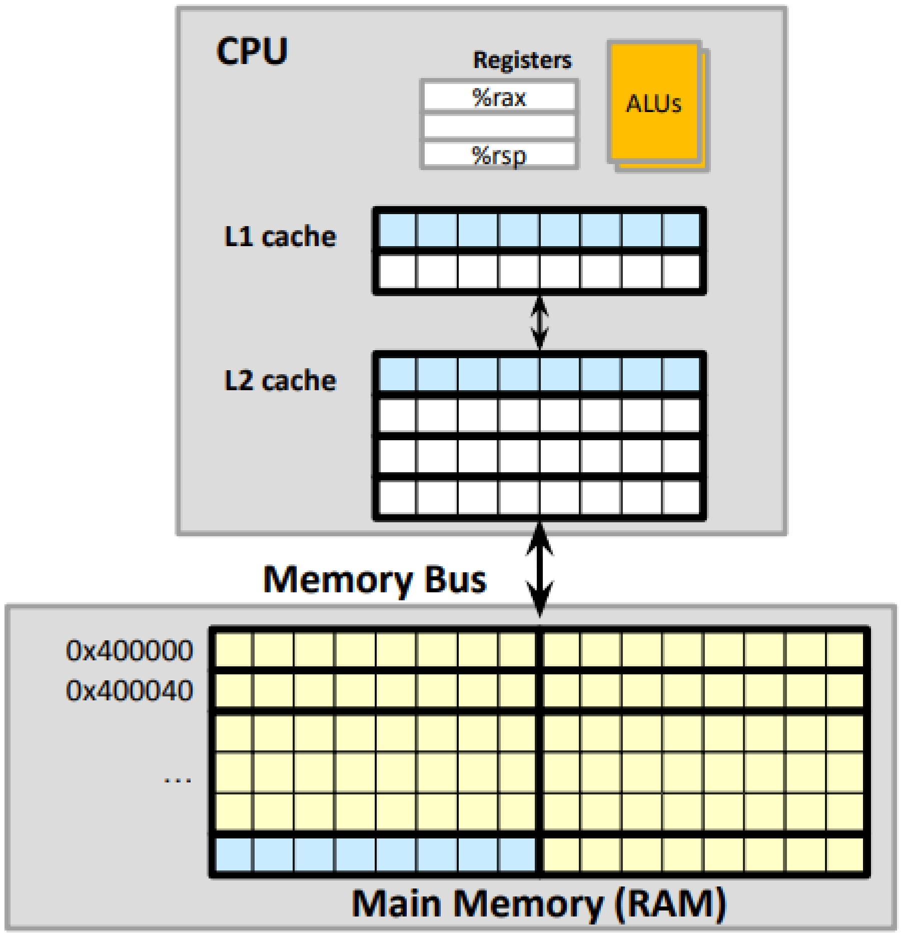
So that leaves us with **load (`L`), store (`S`), and modify (`M`).** A load means we are pulling something from memory (doing something with the variable `var)` , a store means we are putting something into memory (declaring `int var = 2)` , and a modify means we are changing the data in memory (ex. `var` was 2, and we executed `var = 5`). With this in mind, we should **treat load and store the same way (1 access to the address), and modify as load immediately followed by store (2 accesses to the same address).**

An important feature of our cache is that all data is being stored in the cache for some amount of time.

This brings us to the next part of the memory trace: **the address**.

Q: Why do the memory traces specify addresses when we are storing data in the cache?

A: We are **making a copy of a line from main memory** and storing it in our cache. Then, when we are done using that data and need to evict it from the cache, we **propagate the changes into main memory.**



^Here is an illustration of how a line from memory can be moved up and down between caches -- we just have to make sure to write the changes back to main memory when we are all done (Unit 10 Slide 9).

Finally, we have the **size -- number of bytes we are accessing.** As far as the code goes, don't worry about this until you have passed the first two tests.

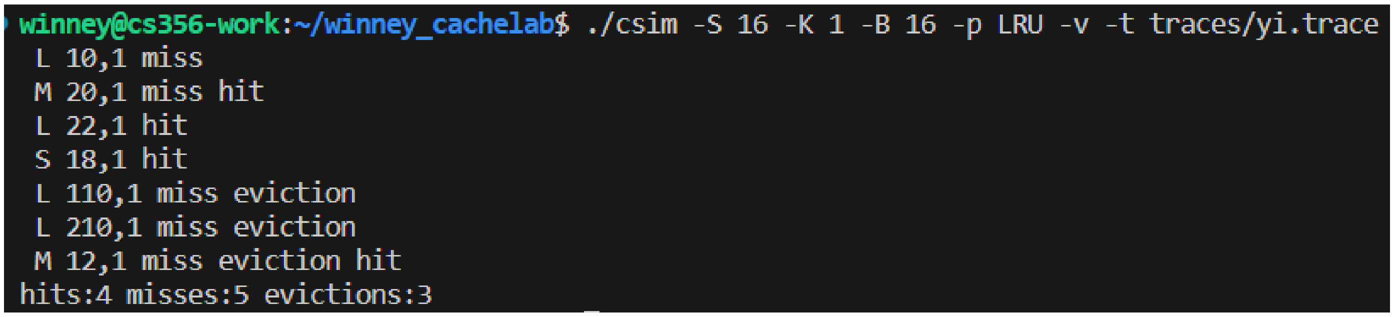
Q: What if the address specifies the **very last byte in a line** (via the block offset), but wants to **access more than one byte?**
A: This is what **crossing a line boundary** is referring to. In this case, because each line of memory accessed goes into the cache, **we need to access the next line in memory too.** Or, if we are **accessing 0x100 bytes and we have** `B = 0x10` bytes per line, we would need to access either **0x10 or 0x11 lines**, depending on **whether we start at the beginning or somewhere in the middle** for the first access. I used hex digits so that the numbers would still be clean, but it wouldn't go against the rule that `B` must be a power of 2.

How to use GDB to debug

IMPLEMENT VERBOSE AS YOU GO. Please. It's easy, and it is how you debug. I cannot help you debug unless you have implemented verbose. If you're having issues with the newlines, remember that there are newlines in the text file. Take that into account.

The trick to debugging is figuring out exactly which memory trace is wrong. You are given the correctly implemented program `csim-ref` which has the verbose option, so you can compare line by line with your verbose output, and if any line is different, your program is wrong and you **don't need to look at any traces after the incorrect trace to debug**.

Really, there is no way to correctly debug. It's mostly using problem solving skills. If one of the traces is wrong, draw out your cache after each memory trace, ie. **fill in all the information from the picture in "The components of the cache."** Then, you can use GDB to see what went wrong.



```
winney@cs356-work:~/winney_cachelab$ ./csim -S 16 -K 1 -B 16 -p LRU -v -t traces/yi.trace
 L 10,1 miss
 M 20,1 miss hit
 L 22,1 hit
 S 18,1 hit
 L 110,1 miss eviction
 L 210,1 miss eviction
 M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

^Here is an example of a program run. It is correct but lets pretend that it's not, and lets **pretend that the fourth line should NOT be `S 18,1` hit.** So, I would draw out my cache through the first few traces, so I could compare it to the state of my cache at the incorrect instruction. Let's do that.

S = 16 --> 16 sets, 4 set bits

B = 16 --> 16 bytes per line, 4 block offset bits

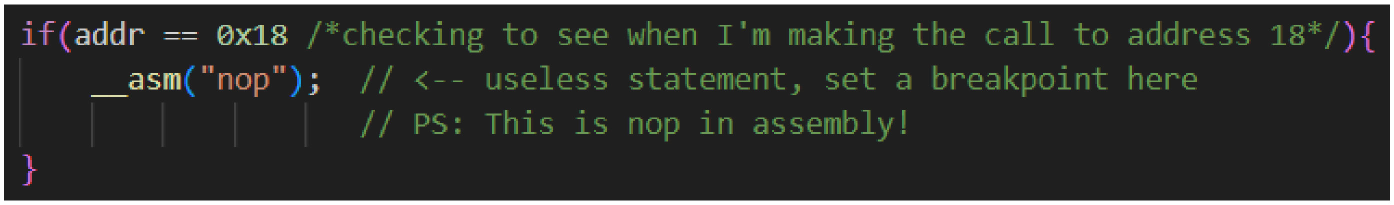
So, to interpret the address, first convert it to binary, and the least significant 4 bits are the block offset bits (ignore), followed by the set bits (the set we need to look in), and the rest are the tag bits (the 'identifier' for the line -- if we have the same tag as one of the lines in the set and the valid bit = 1 for that line, we have a hit).

Here, we only have 1 line, but when we don't have `K = 1,` you may have to scroll through the lines in the set. Likely, your code will start with filling in the first line, then the second line, etc. in each set, so your drawing can match your code in that way too. And once all the lines are filled in, you should know which line is getting evicted or hit or whatever.

L 10,1 miss
--> 0x10 = 0 0001 0000 --> set = 1, tag = 0,

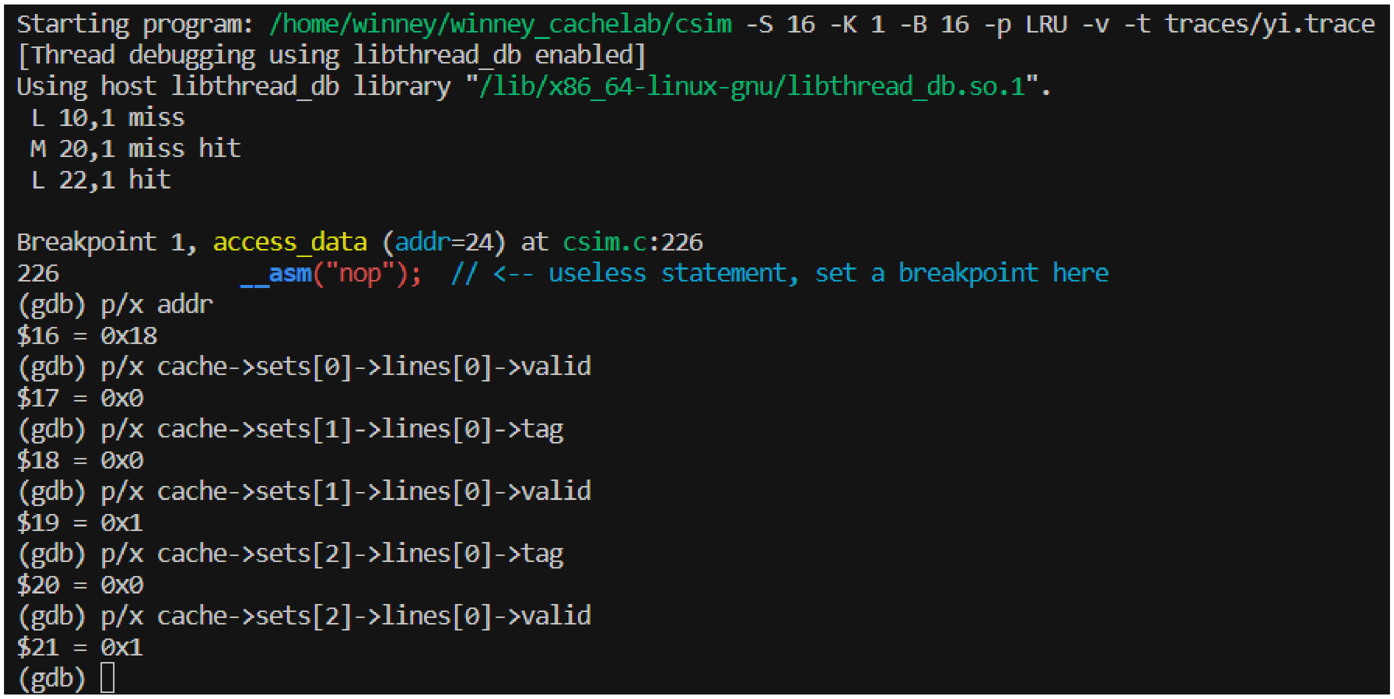
M 20,1 miss hit
--> 0x20 = 0 0010 0000 --> set = 2, tag = 0

L 22,1 hit
--> 0x22 = 0 0010 0010 --> set = 2, tag = 0 --> already have this so we get a hit



```
if(addr == 0x18 /*checking to see when I'm making the call to address 18*/){
    __asm("nop");  // <-- useless statement, set a breakpoint here
                   // PS: This is nop in assembly!
}
```

^I put this at the beginning of `access_data()` . Here we can stop the program when we get to `addr = 0x18` and make sure our cache looks good.



```
Starting program: /home/winney/winney_cachelab/csim -S 16 -K 1 -B 16 -p LRU -v -t traces/yi.trace
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
 L 10,1 miss
 M 20,1 miss hit
 L 22,1 hit

Breakpoint 1, access_data (addr=24) at csim.c:226
226             __asm("nop");  // <-- useless statement, set a breakpoint here
(gdb) p/x addr
$16 = 0x18
(gdb) p/x cache->sets[0]->lines[0]->valid
$17 = 0x0
(gdb) p/x cache->sets[1]->lines[0]->tag
$18 = 0x0
(gdb) p/x cache->sets[1]->lines[0]->valid
$19 = 0x1
(gdb) p/x cache->sets[2]->lines[0]->tag
$20 = 0x0
(gdb) p/x cache->sets[2]->lines[0]->valid
$21 = 0x1
(gdb) 
```

^Here we stopped at our breakpoint, and printed out the contents of set 0, set 1, and set 2 (only has 1 line per set). Does this match the cache that we stepped through by hand?

And you can also step through the code as usual, of course, using "next" or "n."