ECE/CS 250 – Prof. Bletsch
Recitation #7
Caches and Memory

Objective: In this recitation, you will gain a greater understanding of the importance of caches/memory and how they work. This will improve your understanding of caching and virtual memory as well as better prepare you for Homework 5.

Complete as much of this as you can during recitation. If you run out of time, please complete the rest at home.

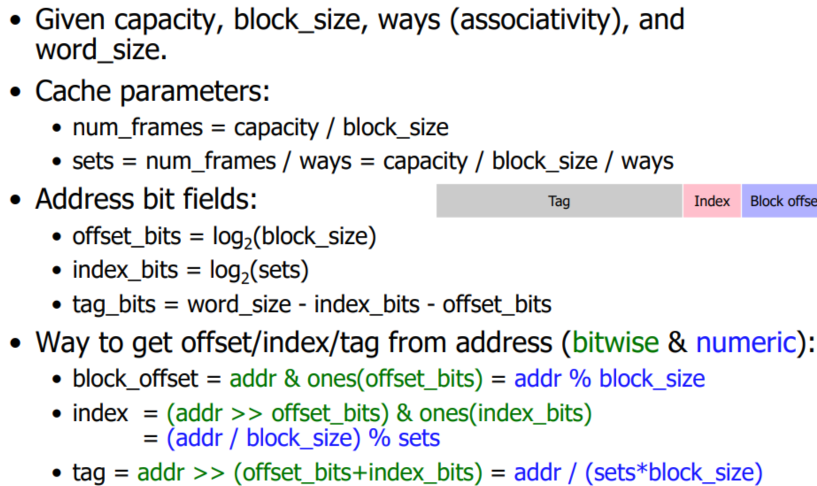
PART 1

1. Cache Examples

You have a 64-bit machine with a cache that is 128KB and 2-way set-associative. Blocks are 64B.

1) How many frames does it have? How many sets does it have? Sketch this cache.
2) Divide up the 64-bit address into its three fields, for purposes of accessing this cache. How many block offset bits are there? How many set index bits are there? How many tag bits are there?
3) For a given sequence of requests, can you explain which ones hit and which ones miss? And for the misses, can you classify them as cold, capacity, or conflict misses? For example, consider the address sequence (assume these are all caused by load-byte instructions): 67, 125, 18, 10, 64K+67, 128K+67.
4) What are the first three blocks that map to set 2? What are the first three blocks that map to set 7? Think about this in two ways:
   a. You know that set 0 hold the blocks [0:63], [64K:64K+63], [128K:128K+63], etc.
   b. You know that all blocks that map to a set have the same set index bits. Write out the set index bits and set the block offset bits to zero (to get the address of the 0th byte in the block). If you set the tag bits to zero, you get the 0th block that maps to this set. If you set the tag bits to 0000…..0001, you get the 1st block that maps to this set. Etc.

Hint: for this task, recall this slide summarizing the cache arithmetic:



2. Virtual Memory Examples

You have a 64-bit machine with 32KB pages. You have 2GB of physical memory.

Key insight: You have as many virtual addresses as $2^{wordsize}$, which constrains how many *virtual* pages there are. Further, you have as many physical addresses as you have physical memory, which constrains how many *physical* pages there are.

(a) How many virtual pages does each process have?

(b) How many physical pages do you have?

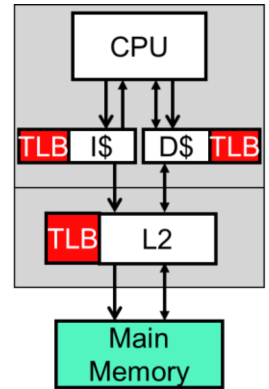
(c) In a flat page table, how many page table entries (PTEs) do you have?

(d) If each PTE is 4 bytes, how big would a "flat" page table be (like the simple lists we saw in class)? This number is crazy big, right? On Homework 5, you'll explore alternative ways large page tables are actually stored instead of a big flat list.

(e) Assume that the page replacement algorithm is LRU. How many distinct virtual pages would you have to see between accesses to a given page X for the second access to page X to be a miss?

3. Caches and Memory – Putting it All Together

(a) To the right is a sketch of a complete memory system that includes the following: L1 I$ and L1 I-TLB, L1 D$ and L1 D-TLB, unified L2$ and L2 TLB, and main memory. Here are many of the possible outcomes for a given load. For each one, explain how it can occur and what happens in the given situation. Why is the shaded situation impossible?



situation	L1 D-TLB	L1 D$	L2 TLB	L2$	mem
1	hit	hit			
2	hit	miss		hit	
3	miss	hit			
4	miss	miss	hit	hit	
5	miss	miss	miss	hit	
6	miss	miss	miss	miss	hit
7	miss	miss	hit	miss	hit
8	miss	miss	hit	miss	miss
9	miss	miss	miss	miss	miss

(b) Let's say you want to implement a virtual/physical cache (virtually indexed and physically tagged, like we saw in class). Assume: 32-bit architecture, 16B cache blocks, 32KB pages. The cache is 128KB. How set-associative must the cache be to permit the use of virtual indexing with physical tagging?

<u>END OF PART 1</u>

If you're in recitation now, please work on the current homework. If it's 100% done, you can head out.

PART 2

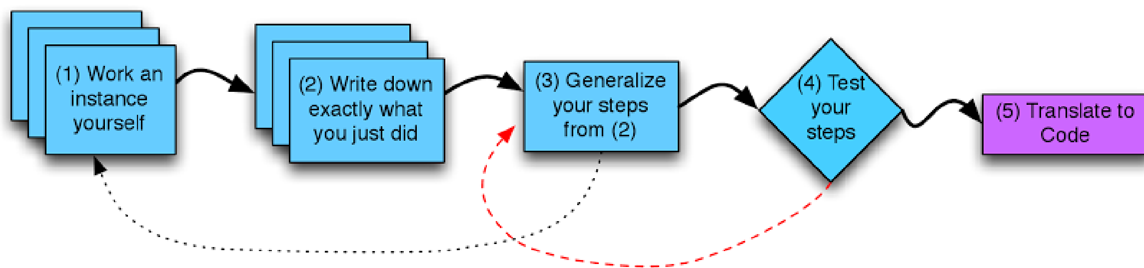
4. Homework 5: Getting the math right

The following steps aren't large; don't overthink them.

(a) How can we use bitshift operations to compute powers of 2? Test your approach in C.
(b) Implement and test the ones() function described in Homework 5.
(c) Write a C function that will give you the lower N bits of a provided 32-bit unsigned integer. Test it.
(d) Write a C function that will get rid of the lower N bits of a provided 32-bit unsigned integer. Test it.
(e) Write a C function that will preserve the lower N bits of a 32-bit unsigned integer X, but replace the upper bits with an input Y. Test it.

5. Homework 5: The virt2phys program structure

The virt2phys program isn't meant to be that long or complex, but it could be if you don't plan appropriately. Below is Prof. Hilton's recommended procedure for designing *any* software:



The "Hilton Method" for algorithm design

Do steps 1-4 for virt2phys program. Don't produce any code, just algorithm notes.

6. Homework 5: Generating cachesim tests

Based on the input and output formats described in the Homework 5 writeup, develop a test trace with at least 10 operations. Then, by hand, generate the corresponding expected simulator output for each of the following cache configurations:

- 4kB cache, 2-way, 8-byte blocks
- 4kB cache, 1-way, 8-byte blocks
- 4kB cache, 2-way, 32-byte blocks

Make sure that your trace file will generate hits, compulsory misses, and conflict misses.

7. Homework 5: Input parsing for cachesim

In C, develop code that can parse the trace file format and print the following information for each input line:

- A Boolean indicating if the line is a "store" operation
- The memory address involved
- The number of bytes in the operation
- For stores, the actual bytes being stored (don't just parrot the hex string – parse it into an array of bytes). Note that you can use `fscanf` with the `%2hhx` specifier to help here.

8. Homework 5: Associativity for cachesim

Develop a data structure and accessor/mutator functions that can associate numeric tags with numeric records. This structure cannot be an array *indexed* by tag, as the tag could potentially be quite large. However, performance isn't a huge concern, so this could be an array of tag/value pairs which are linearly searched. This code will come in handy when implementing associativity in your cache simulator. (Note: if this gets problematic, try using the Hilton Method described earlier.)

ALL DONE?

Nice! Don't head out, though. Work on the current homework and talk to the TAs for help. If you're good on the homework, dig into the experiments below.

Interesting experiments to do if time allows

---

9. Running out of Virtual Memory

On a 64-bit machine, it might appear you could never possibly run out of virtual memory. But what happens if you write a recursive program that (due to a bug) never reaches its base case? Try this. What happens?

10. Running out of Physical Memory

You can never actually run out of physical memory, but you can try to use more memory than you have, in which case the computer spends a lot of its time paging (servicing page faults). Open up the Task Manager to see how much memory you're currently using. Now start launching programs that use a lot of memory. What do you see in the Task Manager?

11. Caches and Memory in the Real World

1) How much cache is in your laptop? You may need to google your CPU model number to find out. To find your CPU model:
    a. Windows: open up the Start menu (bottom right – Windows logo), then right click on the “Computer” option, then choose “Properties”.
        - On Windows, you can also check using a program called CPU-Z, which will directly interrogate the processor and find the amount of cache.
    b. Mac: go to “Apple Logo” > System Report.
    c. Linux: do “cat /proc/cpuinfo” at the command line.
2) How much physical memory do you have? On a Windows machine, you can hit ctrl-alt-delete and then from there open up the Task Manager; in the Task Manager, click on the “Performance” tab. On a Linux machine, you can type “top” at the command line and see much the same information. Once again, I recommend looking at both Windows and Linux.
3) How much disk space do you have? Can you see how much is being used for “swap” (i.e., to hold pages that don’t currently fit in physical memory? This is an aspect of “virtual memory”, which we’ll cover next.

12. Benchmarking memory and cache

1) Download the program “memdance.c” from the course site and compile it with:
    g++ -O3 -o memdance memdance.c
    This program is a simple benchmark to measure the rate of random memory access to a block of memory of a given size for a given duration of a time. Run it without arguments for help.
2) Use this program on the machine of your choice to measure the memory throughput for different buffer sizes. Just run it as “./memdance default”, and it will cycle through buffer sizes of 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, and 128MB for 3 seconds per test.
3) Compare the results to the cache size found in Task 1, especially the lowest level cache (L3 on most modern systems). What do you observe? What’s the percentage difference between the fastest and slowest rate? What does this tell you about the importance of cache to software performance?