Homework #7 – Caching and Virtual Memory

START NOW!

Due Friday, April 21 at 5:00pm

Directions:

- For short-answer questions, submit your answers in PDF format to the GradeScope assignment "Homework 7 written".
- For the programming question, submit your source file using the filename specified in the question to the GradeScope assignment "Homework 7 code".
    - Programs that show good faith effort will receive a minimum of 25% credit.
- **You must do all work individually, and you must submit your work electronically via GradeScope.**
    - All submitted code will be tested for suspicious similarities to other code, and the test will uncover cheating, even if it is "hidden" (by reordering code, by renaming variables, etc.).

Q1. Cache policies

[5 points] Why are write-back caches usually also write-allocate?

Q2. Cache performance

[5] Your L1 data cache has an access latency of 2ns, and your L2 cache has an access latency of 20ns. Assume that 90% of your L1 accesses are hits, and assume that 100% of your L2 accesses are hits. What is the average memory latency as seen by the processor core? **You must show your work to receive full credit.**

Q3. Virtual memory layout

[20] You have a 64-bit machine and you bought 16GB of physical memory. Pages are 512KB. For all calculations, **you must show your work to receive full credit**.

(a) [1] How many virtual pages do you have per process?
(b) [1] How many bits are needed for the Virtual Page Number (VPN)? (Hint: use part (a))
(c) [1] How many physical pages do you have?
(d) [1] How many bits are needed for the Physical Page Number (PPN)? (Hint: use part (c))
(e) [1] How big in *bytes* does a page table entry (PTE) need to be to hold a single PPN plus a valid bit?
(f) [1] How big would a flat page table be for a single process, assuming PTEs are the size computed in part (e)?
(g) [10] Why does the answer above suggest that a “flat page table” isn’t going to work for a 64-bit system like this? Research the concept of a *multi-level page table*, and briefly define it here. Why could such a data structure be much smaller than a flat page table?
(h) [4] Does a TLB miss always lead to a page fault? Why or why not?

Q4. Cache simulator program

[70] In **Java or C**, write a simulator of a single-level cache and the memory underneath it.

The provided test kit uses Java by default. If you write your program in C, you will need to edit `tests/settings.json` to have `“mode”: “exe”.`

The simulator, called `cachesim`, takes the following input parameters on the command line: name of the file holding the loads and stores, cache size (not including tags or valid bits) in kB, associativity, and the block size in bytes. The replacement policy is always LRU. For example, “`cachesim tracefile 1024 4 32`” should simulate a cache that is 1024kB (=1MB), 4-way set-associative, has 32-byte blocks, and uses LRU replacement. This cache will be processing the loads and stores in the file called tracefile.

Important Assumptions: Addresses are 24-bits (3 bytes), and thus addresses range from 0 to 2^{24}-1 (i.e., there is 16MB of address space). The machine is byte-addressed and big-endian. The cache size, associativity, block size, and access size will all be powers of 2. Cache size will be no larger than 2MB, block size will be no larger than 1024 bytes, and no access will be larger than the block size. No cache access will span multiple blocks (i.e., each cache access fits within a single block).

All cache blocks are initially invalid. All cache misses are satisfied by the main memory (and you must track the values written through to memory in case they are subsequently loaded). If a block has never been written before, then its value in main memory is zero. The cache is **write-through** and **write-no-allocate**. This means your program will need to store both the state of cache and the entire content of simulated memory; the memory part can be represented as a simple array of 16M bytes.

If you have any known bugs, please include those in a README file to help the grader give partial credit.

Calling syntax

The program will be called `cachesim`, and will have the following calling syntax:

```
./cachesim <trace-file> <cache-size-kB> <associativity> <block-size>
```

Arguments:

- `<trace-file>`: Filename of the memory access trace file.
- `<cache-size-kB>`: Total capacity of the cache, kilobytes (kB). A power of two between 1 and 2048.
- `<associativity>`: The set associativity of the cache, AKA the number of ways. A power of two.
- `<block-size>`: The size of the cache blocks, in bytes. A power of two between 2 and 1024.

All numeric arguments are in decimal format.

Trace file format

The trace file will be in the following format. There will be some number of lines. Each line will specify a single load or store, the 24-bit address that is being accessed (in base-16), the size of the access in bytes, and the value to be written if the access is a store (in base-16). For example:

```
store 0xd53170 4 7d2f13ac
load  0xd53172 1
store 0xd53170 2 f0b1
store 0x1a25bb 2 c77a
load  0xd53170 4
load  0x12 2
store 0x23 8 d687eb9f1bc687ec
```

As can be seen, leading 0 bits will not be in addresses in the trace file. Also, as viewed in the store commands, values following the access size in bytes will be the correct size. Accesses will be no larger than 8 bytes at a time. Because all parts of the file are whitespace-delimited tokens, `fscanf` will be your friend.

Program output

Your simulator must produce the following output. For every access, it must print out what kind of access it is (load or store), what address it's accessing (in base-16), and whether it is a hit or a miss. For each load, it must print out the value that is loaded (possibly after satisfying the miss from memory). The output format must be as follows and may not be graded if format is ignored. Below is output for the example input file shown above with a 1MB 4-way cache with 32-byte blocks. (Thanks for reading thoroughly; put a picture of a cat in your Q1 answer for extra credit.) This means the cache has 1MB/32 = 32768 frames spread among 4 ways per set, so 32768/4 = 8192 sets. Each line of output is annotated with an explanation:

```
$ ./cachesim traces/example.txt 1024 4 32
store 0xd53170 miss              (First seen, and write-no-allocate means we do NOT cache it now)
load 0xd53172 miss 13            (First *load*, so we DO cache it now; data is based on store above)
store 0xd53170 hit               (We just cached this guy, so store hit)
store 0x1a25bb miss              (First seen, and write-no-allocate means we do NOT cache it now)
load 0xd53170 hit f0b113ac       (Another hit, and see how the data reflects the stores above)
load 0x12 miss 0000              (First load, so we cache it now; data is the starting 00's)
store 0x23 miss                  (First seen also, as this is in a different cache block than 0x12)
```

The "0x" before each address is required; leading 0s are not printed.

Always print out the exact number of hex digits corresponding with the number of bytes missed. Memory defaults to all zeroes on boot.

Restriction

In this assignment, **you may NOT use the modulus (%) operator** and **you may NOT include math.h, java.lang.Math, or otherwise use built-in modulus, logarithmic, or power functions**. You must use bitwise operations to decompose the address. Penalty: 50% off overall score.

Additionally, your program should exit with a status of 0 (EXIT_SUCCESS). Penalty: 25% of score.

You do NOT have to worry about memory leaks on this assignment. However, valgrind is still a useful tool for detecting misuse of memory, and may help you find hidden bugs!

Building and testing

For C, you can simply build your program as per usual with g++:

```
g++ -g -o cachesim cachesim.c
```

For Java, you can run `javac` on your program:

```
javac cachesim.java
```

A number of input files are in the `traces` subdirectory; these are described by `traces/INFO.txt`. As with prior assignments, a suite of tests is provided in the `tests` subdirectory and an automated testing tool, `hwtest.py`, has been provided to automate testing.

Tips relevant to Q4

- To manipulate bit fields, use bitwise operators (`&`, `|`, `~`), shifts (`<<`, `>>`) and masks.
- You can compute 2^N with the expression: `(1<<N)`
- You can get a bit string of N ones with the expression: `((1<<N)-1)`
- Here's a simple implementation of base-2 log using only integer math, that way you don't have to mess with the math library:

```
int log2(int n) {
   int r=0;
   while (n>>=1) r++;
   return r;
}
```

- Parsing/printing tips for `cachesim`:
  - You can `fscanf` two hex digits at a time using the "`%2hhx`" specifier.
  - Similarly, you can print a zero-padded two-digit hex representation of a byte with the "`%02hhx`" `printf` specifier.
  - You can provide known parts of the input format in the `fscanf` format, such as the "0x" part of the hex address for `cachesim`.
  - See the manpages for `scanf` and `printf` for more information.
  - `BigInteger` in Java lets you have arbitrary-length integers if you want to avoid manually manipulating strings or arrays.