CSE 361 Fall 2015
Cache Lab: Building a Cache Simulator
Assigned: Wednesday, October 7, 2015
Due: Friday October 23 at 11:59 pm

This lab will help you understand how cache memory works. You will write a small C program (about 300 lines) that simulates the behavior of a cache memory.

Logistics

This is an individual project. All handins are electronic via committing code to your svn repository.

Clarifications and corrections will be posted on the course Piazza page. This pdf will also be updated as needed, so please refresh before you begin working each time.

SVN Instructions

In order to see the new `lab4` directory that we have pushed to your repository, perform an `svn update`.

Inside `lab4` will be a number of files (see README). You will be modifying the file `csim.c`.

Handin Instructions

The file `csim.c` is the *only* file that the autograder will collect. A binary executable `test-csim` has been provided for you to test the correctness of your cache simulator. It is your job to ensure that you have successfully committed your code to the svn repository, and that your code compiles and runs successfully when `test-csim` is invoked. (See the Evaluation section for more detail on using `test-csim`.)

To compile, type:

```
linux> make clean
linux> make
```

NOTE: You will not receive any credit if your code does not compile.

To make sure that you have submitted your code correctly, be sure to check that your most recent changes are committed to the repository by viewing the repo on the web at the URL:

`https://svn.seas.wustl.edu/repositories/<yourwustlkey>/cse361s_fl15/`

(The repo is only viewable by those with permissions)

Building a Cache Simulator

You will write a cache simulator in `csim.c` that takes a `valgrind` memory trace as input, simulates the hit/miss behavior of a cache memory on this trace, and outputs the total number of hits, misses, and evictions.

Reference Trace Files

The `traces` subdirectory of the handout directory contains a collection of *reference trace files* that we will use to evaluate the correctness of the cache simulator you write.

The trace files are generated by a Linux program called `valgrind`. For example, typing

```
linux> valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

on the command line runs the executable program "`ls -l`", captures a trace of each of its memory accesses in the order they occur, and prints them on `stdout`.

`Valgrind` memory traces have the following form:

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

Each line denotes one or two memory accesses. The format of each line is

```
[space]operation address,size
```

The *operation* field denotes the type of memory access: "I" denotes an instruction load, "L" a data load, "S" a data store, and "M" a data modify (i.e., a data load followed by a data store). There is never a space before each "I". There is always a space before each "M", "L", and "S". The *address* field specifies a 64-bit hexadecimal memory address. The *size* field specifies the number of bytes accessed by the operation.

Reference Cache Simulator

We have provided you with the binary executable of a *reference cache simulator*, called `csim-ref`, that simulates the behavior of a cache with arbitrary size and associativity on a `valgrind` trace file. It uses the LRU (least-recently used) replacement policy when choosing which cache line to evict.

The reference simulator takes the following command-line arguments:

```
Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

- `-h`: Optional help flag that prints usage info
- `-v`: Optional verbose flag that displays trace info
- `-s <s>`: Number of set index bits ($S = 2^s$ is the number of sets)
- `-E <E>`: Associativity (number of lines per set)
- `-b <b>`: Number of block bits ($B = 2^b$ is the block size)
- `-t <tracefile>`: Name of the `valgrind` trace to replay

The command-line arguments are based on the notation (s, E, and b) from section 6.4 of the CS:APP2e textbook. For example:

```
linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

The same example in verbose mode:

```
linux> ./csim-ref -v -s 4 -E 1 -b 4 -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
```

Your job is to fill in the `csim.c` file so that it takes the same command line arguments and produces the identical output as the reference simulator. Notice that this file is almost completely empty. You'll need to write it from scratch.

Programming Rules

- Include your name and WUSTL key in the header comment for `csim.c`.
- Your `csim.c` file must compile without warnings in order to receive credit.
- Your simulator must work correctly for arbitrary s, E, and b. This means that you will need to allocate storage for your simulator's data structures using the `malloc` function. Type "man malloc" for information about this function.

- For this lab, we are interested only in data cache performance, so your simulator should ignore all instruction cache accesses (lines starting with "I"). Recall that `valgrind` always puts "I" in the first column (with no preceding space), and "M", "L", and "S" in the second column (with a preceding space). This may help you parse the trace.
- To receive any credit, you must call the function `printSummary`, with the total number of hits, misses, and evictions, at the end of your `main` function:

```
printSummary(hit_count, miss_count, eviction_count);
```

- For this this lab, you should assume that memory accesses are aligned properly, such that a single memory access never crosses block boundaries. By making this assumption, you can ignore the request sizes in the `valgrind` traces.

Evaluation

This section describes how your work will be evaluated. The full score for this lab is 33 points.

We will run your cache simulator using different cache parameters and traces. The short traces worth 3 points each, and the long traces (last two shown) worth 6 points each. The `test-csim` program runs your simulator against 9 test cases with the given traces:

```
linux> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
linux> ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
linux> ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
linux> ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
linux> ./csim -s 6 -E 8 -b 6 -t traces/sort.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/long.trace
```

Note that, **when we autograde test your code, we will run your simulator using different cache parameters for a subset of these tests and on one additional hidden trace**, so you should be sure that your simulator is coded to work for arbitrary cache parameters. If your simulator pass the `test-csim` tests and is coded to handle arbitrary cache parameters, then your code will likely work for any test cases that we use for grading.

You can use the reference simulator `csim-ref` to obtain the correct answer for different test cases. During debugging, use the `-v` option for a detailed record of each hit and miss.

For each test case, outputting the correct number of cache hits, misses and evictions will give you full credit for that test case. Each of your reported number of hits, misses and evictions is worth 1/3 of the credit for that test case. That is, if a particular test case is worth 3 points, and your simulator outputs the correct number of hits and misses, but reports the wrong number of evictions, then you will earn 2 points.

Testing Your Code

We have provided you with an autograding program, called `test-csim`, that tests the correctness of your cache simulator on the reference traces with a subset of test cases. Be sure to compile your simulator before running the test:

```
linux> make
linux> ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (6,8,6)  141650    4450    3938  141650    4450    3938  traces/sort.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    33
```

For each test, it shows the number of points you earned, the cache parameters, the input trace file, and a comparison of the results from your simulator and the reference simulator.

NOTE: We will use a similar program to test your code on linuxlab machines, so be sure to test your code on linuxlab machines before submission!

Hints and Suggestions

Here are some hints and suggestions:

- Do your initial debugging on the small traces, such as `traces/dave.trace`.
- The reference simulator takes an optional `-v` argument that enables verbose output, displaying the hits, misses, and evictions that occur as a result of each memory access. You are not required to implement this feature in your `csim.c` code, but we strongly recommend that you do so. It will help you debug by allowing you to directly compare the behavior of your simulator with the reference simulator on the reference trace files.
- We recommend that you use the `getopt` function to parse your command line arguments. You'll need the following header files:

  ```
  #include <getopt.h>
  #include <stdlib.h>
  #include <unistd.h>
  ```

  See "`man 3 getopt`" for details.

- Each data load (L) or store (S) operation can cause at most one cache miss. The data modify operation (M) is treated as a load followed by a store to the same address. Thus, an M operation can result in two cache hits, or a miss and a hit plus a possible eviction.
- You are strongly encouraged to create your own trace file for testing using `valgrind`:

  ```
  linux> valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes <program>
  ```

  `Valgrind` outputs some extra lines at the beginning and end of the the trace (line starts with ==) that you may want to remove. Your simulator is not expected to handle these extra output.