CPTS 360: Programming Assignment 1

Cache Memory Simulation

Notes:

- This is an individual assignment.
- You should complete the lab on a Linux environment (either a dedicated machine or VM). Using WSL (Windows Subsystem for Linux) on Windows is NOT recommended.
- Your submission will be tested on a Linux environment. You will NOT receive any points if your submitted program does not work on a Linux system.
- Start EARLY. The lab may take more time than you anticipated.
- Read the entire document carefully before you start working on the lab.
- The code and other answers you submit MUST be entirely your own work, and you are bound by the WSU Academic Integrity Policy (`https://www.communitystandards.wsu.edu/policies-and-reporting/academic-integrity-policy/`).
- You MAY consult with other students about the conceptualization of the tasks and the meaning of the questions, but you MUST NOT look at any part of someone else's solution or collaborate with anyone.
- You may consult published references or search online archives, provided that you appropriately cite them in your reports/programs.
- The use of artificial intelligence (AI)–generated texts/code snippets must be CLEARLY DISCLOSED, including the prompt used to generate the results and the corresponding outputs the tool(s) provide.

Good Luck!

1 Introduction

In this programming assignment, you will write a C program that simulates the behavior of a cache memory. The cache simulator that takes a `valgrind` memory trace as input (see details in Section 2.1), simulates the hit/miss behavior of a cache memory on this trace, and outputs the total number of hits, misses, and evictions.

2 Problem Description

You will be modifying a given (almost empty) C file: `cachesim.c`. To compile it, type:

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

2.1 Reference Trace Files

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

2.2 Developing a Cache Simulator

We assume LRU (least-recently used) replacement policy when choosing which cache line to evict. The simulator takes the following command-line arguments:

```
Usage: ./cachesim [-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) discussed in the class (also used in the CSAPP textbook). For example:

```
linux> ./cachesim -s 4 -E 1 -b 4 -t traces/trace02.dat
hits:4 misses:5 evictions:3
```

The same example in verbose mode:

```
linux> ./cachesim -v -s 4 -E 1 -b 4 -t traces/trace02.dat
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 task is to complete `cachesim.c` file so that it takes the same command line arguments and produces the identical output in the above format.

2.3 Assignment Guidelines

- The simulator source file (`cachesim.c`) file must be compiled by the provided `Makefile` without warnings in order to receive credit.
- Your simulator must work correctly for arbitrary s, E, and b. Hence, you should allocate storage for your simulator's data structures using the `malloc` function. Check "`man malloc`" for information about this function.
- For this assignment, 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 credit, you must call the function `print_summary`, with the total number of hits, misses, and evictions at the end of your `main` function:

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

- The simulator must support the optional `-v` argument that enables verbose output, displaying the hits, misses, and evictions resulting from each memory access. It will also help you debug by allowing you to directly compare the behavior of your simulator with the sample output trace files.
- We recommend that you use the `getopt` function to parse your command line arguments. You will need the following header files:

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

  See "`man 3 getopt`" for details.
- In addition, for invalid input arguments (and for `-h` flag), your code should call the function `print_usage` to display the usage information. For example:

(*s*,*E*,*b*)	Hits	Misses	Evicts	Trace
(1,1,1)	9	8	6	`traces/trace01.dat`
(4,2,4)	4	5	2	`traces/trace02.dat`
(2,1,4)	2	3	1	`traces/trace03.dat`
(5,1,5)	265189	21775	21743	`traces/trace04.dat`

Table 1: Cache Behavior (Expected Outputs).

```
while( (c=getopt(argc,argv,"s:E:b:t:vh")) != -1){
        switch(c){
        // other condition checking
        ...
        case 'h':
            print_usage(argv);
            exit(0);
        default:
            print_usage(argv);
            exit(1);
        }
    }
```

- 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.
- 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.

2.4 Testing and Debugging

We will run your cache simulator using different cache parameters and traces. There are four test cases:

```
linux> ./cachesim -s 1 -E 1 -b 1 -t traces/trace01.dat
linux> ./cachesim -s 4 -E 2 -b 4 -t traces/trace02.dat
linux> ./cachesim -s 2 -E 1 -b 4 -t traces/trace03.dat
linux> ./cachesim -s 5 -E 1 -b 5 -t traces/trace04.dat
```

We provide the expected results for all four test cases; see Table 1. In addition, we also provide sample (verbose) outputs for `trace01` and `trace02` (see `traces/output/` directory).

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 are 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. **Note:** your code must output correctly in the "verbose" mode as well. If the numbers in verbose output are incorrect, you will lose $1/2$ points for that test case.

3 Deliverable

- Complete the implementation of the cache simulator code. You can use multiple source/header (`.c` and `.h`) files (update `Makefile` if you add files). The main file of our implementation (i.e., where the `main()` routine begins) should be `cachesim.c`. **[90 Points]**
  - The point split for your implementation is as follows: **80 points** for correctness, **10 points** for good code organization, indentation, and proper comments.
  - We will test your code by using the four traces we provided (60 out of 80 points; each trace has equal points). The remaining 20 points will be used for testing the correctness and completeness of your code with a different trace file/cache setup, which is not shared with you.
- A PDF document summarizes how you implemented your cache simulator (see below for formatting details). **[10 Points]**
  - Your report should be **no more than two pages**; use standard **letter paper size** ($8^1/_2$ by 11 inches) with **11 points Times font** and **1-inch margin**.
  - Your PDF filename should be `pa1_lastname.pdf`, where `lastname` is your surname.
  - Include your **full name** and **WSU ID** inside the report.
  - Include your report on the GitHub repository.
  - We will **deduct 5 points** if the filename/contents do not follow this convention and/or you use a different font/margin than that listed above.

4 Submission Guidelines

Commit and push your work (source and report) on the GitHub Classroom Programming Assignment 1 repository. **Make sure you can successfully push commits on GitHub Classroom — *last minute excuses will not be considered***. Paste the URL of your GitHub repo to the corresponding lab assignment section of Canvas. Check the course website for additional details.

Note: Your work on GitHub Classroom will not be considered for grading unless you submit the link to the GitHub repository URL on Canvas. Hence, **You will NOT receive any points if you fail to submit your URL on Canvas.** Submissions received after the due date will be graded based on the late submission policy as described in the course syllabus.

Acknowledgement. This assignment was initially developed by the team members of the course textbook (CSAPP) and has been customized for CPTS 360 by the instructor.