Assignment #5: Cache Lab (due on Wed, Apr. 3, 2019 at 11:59pm)

Introduction

This lab will help you understand the impact that cache memories can have on the performance of your C programs. You will write a small C (not C++!) program of about 200-300 lines that simulates the behavior of a cache memory. You will find the starting point in your repository, under the directory `proj5`.

Input Trace Files

The `traces` directory contains a collection of *reference trace files* that we will use as input to evaluate the correctness of your cache simulator. The trace files are generated by a Linux program called `valgrind`. For example, typing:

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

on the command line, `valgrind` 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.

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 an `I` but there is always a space before each `M`, `L`, and `S` (this is important to parse these files).
- The `address` field specifies a 64-bit hexadecimal memory address.
- The `size` field specifies the number of bytes accessed by the operation.

Required Command-Line Interface and Reference Implementation

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.

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 either the LRU (least-recently used) replacement policy or FIFO (First-In First-Out) when choosing which cache line to evict, as per the selected `-L` or `-F` flags, respectively.

The reference simulator takes the following command-line arguments:

```
./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile> (-L|-F)
```

- `-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
- `-L`: Set the cache eviction policy to be LRU.
- `-F`: Set the cache eviction policy to be FIFO.

The command-line arguments are based on the notation (`s`, `E`, and `b`) from page 617 of the CS:APP3e textbook:

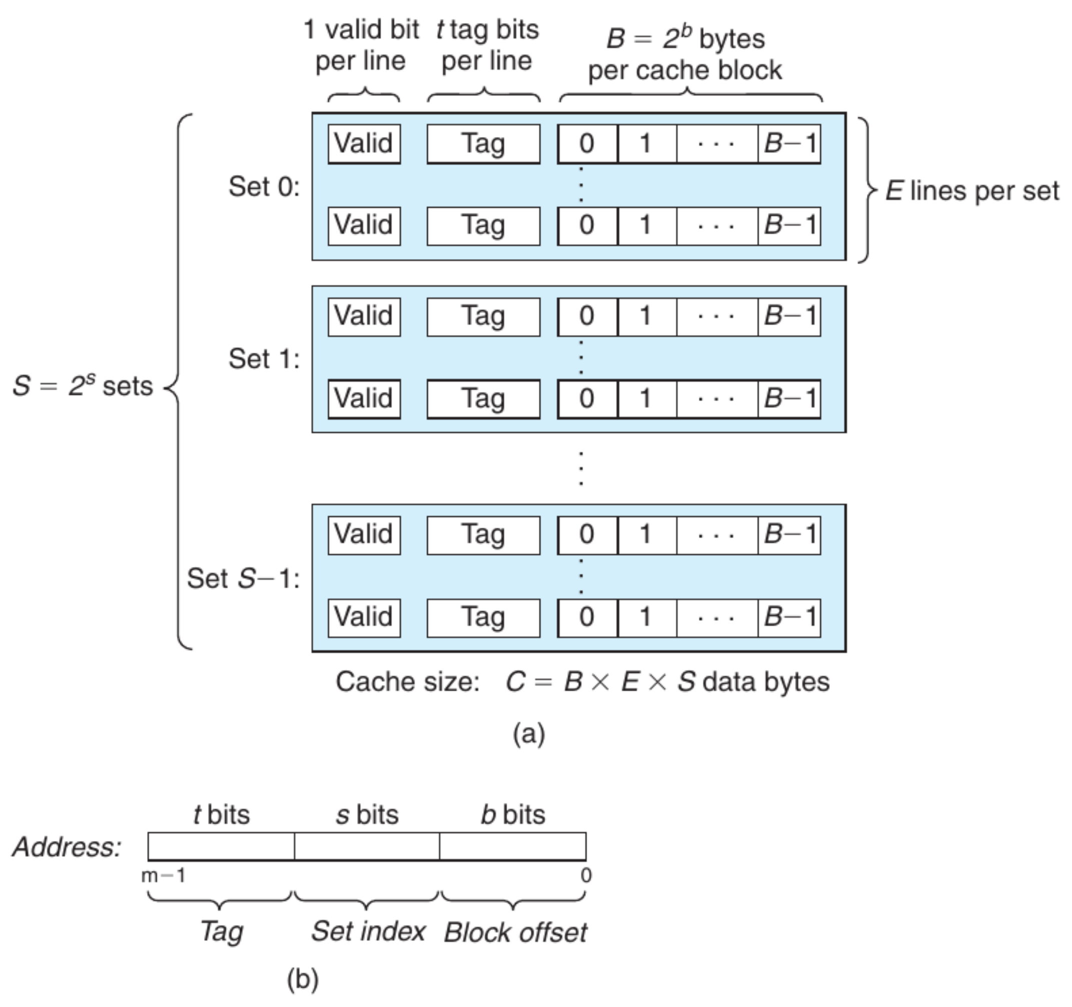

Figure 6.27 **General organization of cache (S, E, B, m).** (a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data. (b) The cache organization induces a partition of the m address bits into t tag bits, s set index bits, and b block offset bits.

For example, you can start the reference cache simulator like this:

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

The same example in verbose mode annotates each memory access of the input trace as `hit`/`miss`:

```
$ ./csim-ref -v -s 4 -E 1 -b 4 -L -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 task for this project 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.

Look at the figure above and think about the data structures that you need!
As you read `L` / `S` / `M` accesses from the input trace, you will need (at least) to:

- split the memory address;
- use the "set" portion to select the correct set (in some data structure);
- use the "tag" portion to search for a line (in some data structure);
- update counters/metadata to keep track of which line is the oldest (for FIFO) or the least recently accessed (for LRU).

Programming Rules

You must adhere to the following rules.

- Include your name and USC username 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.
- Your simulator must accurately use both the FIFO and LRU cache eviction policies, as determined by the command line parameters.

- 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 in parsing the trace.
- To receive credit for this project, 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 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** (i.e., it is never the case that an access involves two consecutive cache lines).
- You may only use C code (no C++) that will compile with `gcc` and the `-std=c99` flags.

Evaluation

We will run your cache simulator using different cache parameters and traces. There are 16 test cases: 14 cases worth 3 points each, and 2 cases worth 6 points each. The full score for this lab is 54 points.

In particular, 8 tests will be run for each cache replacement policy (LRU, FIFO). The LRU test cases are as follows:

```
$ ./csim -s 1 -E 1 -b 1 -L -t traces/yi2.trace
$ ./csim -s 4 -E 2 -b 4 -L -t traces/yi.trace
$ ./csim -s 2 -E 1 -b 4 -L -t traces/dave.trace
$ ./csim -s 2 -E 1 -b 3 -L -t traces/trans.trace
$ ./csim -s 2 -E 2 -b 3 -L -t traces/trans.trace
$ ./csim -s 2 -E 4 -b 3 -L -t traces/trans.trace
$ ./csim -s 5 -E 1 -b 5 -L -t traces/trans.trace
$ ./csim -s 5 -E 1 -b 5 -L -t traces/long.trace
```

The FIFO test cases are as follows:

```
$ ./csim -s 4 -E 2 -b 4 -F -t traces/fifo_s1.trace
$ ./csim -s 4 -E 2 -b 4 -F -t traces/fifo_s2.trace
$ ./csim -s 4 -E 4 -b 4 -F -t traces/fifo_s3.trace
$ ./csim -s 5 -E 2 -b 2 -F -t traces/fifo_m1.trace
$ ./csim -s 3 -E 4 -b 2 -F -t traces/fifo_m1.trace
$ ./csim -s 5 -E 2 -b 2 -F -t traces/fifo_m2.trace
$ ./csim -s 3 -E 4 -b 2 -F -t traces/fifo_m2.trace
$ ./csim -s 4 -E 2 -b 4 -F -t traces/fifo_l.trace
```

You can use the reference simulator `csim-ref` to obtain the correct answer for each of these 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.

Checking Your Score

We have provided you with an autograding program, called `test-csim`, that tests the correctness of your cache simulator on the reference traces. **Be sure to compile your simulator before running the test.** Running `test-csim` on a correct submission would produce the following output:

```
$ make
$ ./test-csim
EP: LRU                 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 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27

EP: FIFO                Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (4,2,4)       7       5       2       7       5       2  traces/fifo_s1.trace
     3 (4,2,4)      11       7       3      11       7       3  traces/fifo_s2.trace
     3 (4,4,4)       6      11       7       6      11       7  traces/fifo_s3.trace
     3 (5,2,2)      59     354     298      59     354     298  traces/fifo_m1.trace
     3 (3,4,2)      51     362     330      51     362     330  traces/fifo_m1.trace
     3 (5,2,2)     191     188     142     191     188     142  traces/fifo_m2.trace
     3 (3,4,2)     164     215     184     164     215     184  traces/fifo_m2.trace
     6 (4,2,4)  263447   28255   28223  263447   28255   28223  traces/fifo_l.trace
    27

TEST_CSIM_RESULTS=54
```

For each test, `test-csim` 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.

Hints

Here are some hints and suggestions for working on this assignment:

- 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.
- You are allowed to search the Internet for the documentation of the following C library functions which will be helpful: `getopt` (to parse command-line arguments), `fopen` and `fclose` (to open/close files), `fgets`, `fscanf`, or `getline` (to read from files), `sscanf` (to parse fields within a string).
- Note that `sscanf` requires the format string to contain the expected format. For example, if you wanted to parse two `int` separated by a comma and a space you would have to use a format string like: `"%d, %d"` that contains the expected placeholders at the expected locations.
- 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.

To get you up to speed with command-line arguments and reading line from files in C, we provide the following example, which you can use as a starting point (again, cite your source!). Note that it doesn't handle all of the required command-line arguments.

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

int main (int argc, char **argv) {

    /* use ':' to read arguments, otherwise it's a flag */
    char *options = "vE:t:";

    /* to check for errors in str to long conversion    */
    char *errpos;

    /* data that we're expecting                        */
    int verbose = 0;          /* from switch -v         */
    char *filename = NULL;    /* from argument of -t    */
    long lines_per_set = 0;   /* from argument of -E    */

    /* getopt returns -1 when parsing is over           */
    /* the parsed arguments are saved in getopt         */
    int opt_char = 0;
    while ((opt_char = getopt(argc, argv, options)) != -1) {
        switch (opt_char) {
        case 'E':
            /* atoi is deprecated, learn strtol instead!!!       */
            errpos = NULL;        /* ptr to end of parsed string */
            lines_per_set = strtol(optarg, &errpos, 10);
            if (*errpos != '\0') {
                fprintf(stderr, "ERROR: Invalid input to -E: %s\n", optarg);
                return 1;
            }
            break;

        case 't':
            filename = optarg;
            break;

        case 'v':
            verbose = 1;
            break;

        case '?':
        default:
            fprintf(stderr, "ERROR: Parsing error in getopt\n");
            return 1;
        }
    }

    for (int i=optind; i < argc; i++) {
        printf("WARNING: Ignored argument '%s'\n", argv[i]);
    }

    if (filename == NULL) {
        fprintf(stderr, "ERROR: Must specify trace file with -t\n");
        return 1;
    }

    if (lines_per_set <= 0) {
        fprintf(stderr, "ERROR: Must specify a positive value for -E\n");
        return 1;
    }

    printf(">> verbose = %d, trace = %s, E = %ld\n",
           verbose, filename, lines_per_set);
    printf(">> Contents of file %s:\n", filename);
    FILE *fp = fopen(filename, "r");
    if (fp == NULL) {
        perror("ERROR"); /* prints message for errno */
        return 1;
    }

    /* if line == NULL && len == 0,
       getline allocates a char array using malloc
       and reallocates that at every call            */
    char *line = NULL;
    size_t len = 0;
    ssize_t nread;
    while ((nread = getline(&line, &len, fp)) != -1) {
        fwrite(line, nread, 1, stdout);
    }

    /* deallocate the last line allocated by getline */
    free(line);

    /* close the file */
    fclose(fp);

    return 0;
}
```

Acknowledgements. This lab was developed by the authors of the course textbook and their staff. It has been customized for use by this course.

CS356