Assignment #5: Cache Lab (due on Wed Nov 13, 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 to 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.

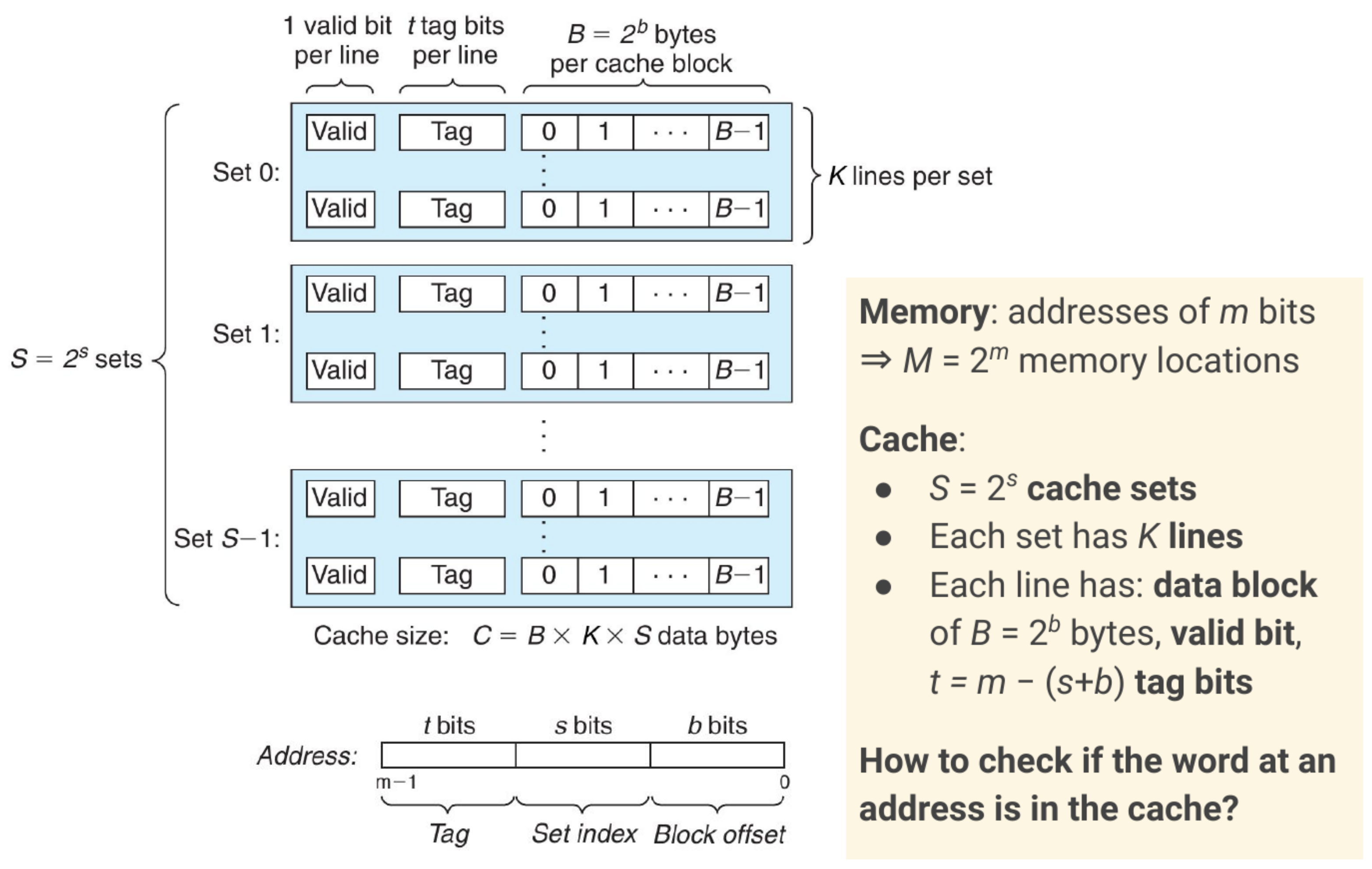
We have provided you with the binary executable of a reference cache simulator, called `csim-ref`, that simulates the behavior of a cache with arbitrary size and associativity on a `valgrind` trace file. It uses either the LRU (least-recently used) replacement policy or FIFO (First-In First-Out) when choosing which cache line to evict, as specified with the `-p LRU` or `-p FIFO` option, respectively.

The reference simulator takes the following command-line arguments:

```
./csim-ref [-hv] -S <S> -K <K> -B <B> -p <P> -t <tracefile>
```

- `-h`: Optional help flag that prints usage info
- `-v`: Optional verbose flag that displays trace info
- `-S <S>`: Number of sets ($s = \log_2(S)$ is the number of bits used for the set index)
- `-K <K>`: Number of lines per set (associativity or number of *ways*)
- `-B <B>`: Number of bytes per line ($b = \log_2(B)$ is the number of bits used for the byte offset within the line)
- `-p <P>`: Selects a policy, either `LRU` or `FIFO`
- `-t <tracefile>`: Name of the valgrind trace to replay

The command-line arguments are based on the notation (`S`, `K`, `B`) from page 617 of the CS:APP3e textbook:



Memory: addresses of m bits
⇒ $M = 2^m$ memory locations

Cache:

- $S = 2^s$ **cache sets**
- Each set has K **lines**
- Each line has: **data block** of $B = 2^b$ bytes, **valid bit**, $t = m - (s+b)$ **tag bits**

How to check if the word at an address is in the cache?

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

```
$ ./csim-ref -S 16 -K 1 -B 16 -p LRU -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 -S 16 -K 1 -B 16 -p LRU -v -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 identical output to the reference simulator.** This file already has some code, but you need to figure out what data structures you need and how to use them! (Look at the figure above.)

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.

- Your `csim.c` file must compile **without warnings in order to receive any credit.**
- Your simulator must work correctly for variable `S`, `K`, 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 `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);`
- You may only use C code (no C++) that will compile with `gcc` and the `-std=c11` flags.

Evaluation

We will run your cache simulator using different test suites:
- **Direct Mapped** (1 point): These tests use only one line per set ($K = 1$), so that all eviction policies are equivalent (you don't even need to implement a policy to pass these tests).
- **Policy Tests** (2 points): These tests check that your LRU and FIFO policies work correctly.
- **Size Tests** (2 points): These tests include memory accesses that can cross a line boundary. For example, *if your cache lines are 4 bytes, reading 4 bytes from `0x2` will generate two memory accesses* (to the block `0x0-0x3` and to the block `0x4-0x7`). For these tests, you need to take into account the `size` field of `valgrind` traces.

You must pass all tests in a test suite to receive its points. You can check your grade with `./grade`:

```
$ ./grade
rm -rf csim
gcc -g -Wall -Werror -std=c11 -o csim csim.c
==
>> Running 'direct' tests ... FAILED
./csim -S 8 -K 1 -B 1 -p LRU -t traces/simple_direct.trace
EXPECTED: hits:3 misses:3 evictions:0
  ACTUAL: hits:2 misses:3 evictions:0
==
>> Running 'policy' tests ... FAILED
./csim -S 2 -K 2 -B 2 -p LRU -t traces/simple_policy.trace
EXPECTED: hits:3 misses:4 evictions:1
  ACTUAL: hits:2 misses:4 evictions:1
==
>> Running 'size' tests ... FAILED
./csim -S 1 -K 1 -B 2 -p LRU -t traces/simple_size.trace
EXPECTED: hits:2 misses:6 evictions:5
  ACTUAL: hits:2 misses:6 evictions:4
==
>> SCORE: 0
```

If a test is not passing, **you can use the reference simulator `csim-ref` with the `-v` option to obtain a detailed record of each hit, miss, eviction:**

```
$ ./csim-ref -S 8 -K 1 -B 1 -p LRU -v -t traces/simple_direct.trace
L 0,1 miss
L 2,1 miss
L 0,1 hit
L 2,1 hit
L 4,1 miss
L 0,1 hit
hits:3 misses:3 evictions:0
```

Hand-In Instructions

You must commit and push your `csim.c` file to GitHub. Assignment collection will be automatic: after the assignment deadline, our grading system will fetch the most recent commit on the `master` branch of your repository.

Be sure to run `./grade` and verify your score.

If you want to use late days, make a push after the deadline: we will use the push date (not the commit date) to determine your late days.

Hints

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

- Do your initial debugging on the simple traces (`traces/simple_*`).
- 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.
- 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 (even when only 1 byte is modified).

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