p4B: Coding a Cache Simulator

Due Apr 7 by 10pm **Points** 80 **Submitting** a file upload
Available until Apr 7 at 10:33pm

This assignment was locked Apr 7 at 10:33pm.

- This project must be done individually.
  - *It is academic misconduct to share your work with others in any form including posting it on publicly accessible web sites, such as GitHub.*
  - *It is academic misconduct for you to copy or use some or all of a program that has been written by someone else.*
- All work for this project is to be done on the **CS Department's instructional Linux machines (https://csl.cs.wisc.edu/services/instructional-facilities)** . You're encouraged to remotely login using the ssh command in the terminal app on Macs, or on Windows using an ssh client such as MobaXterm.
- All projects in this course are graded on **CS Department's instructional Linux machines (https://csl.cs.wisc.edu/services/instructional-facilities)** . To receive credit, make sure your code runs as you expect on these machines.

Learning GOALS

In part A of this assignment, you learned how to use a cache simulator to make inferences about caches with different parameters. In this part, you'll develop your own basic cache simulator. This will further your understanding of cache basics and strengthen your C programming skills.

Getting the FILES

Copy the compressed file `/p/course/cs354-skrentny/public/code/p4B/p4Bfiles.tgz` to your private **cs354** working directory. Next enter the following command inside that directory:

```
[skrentny@jimbo] (44)$ tar -zxvf p4Bfiles.tgz
```

This creates a subdirectory named "**p4B**" with the required files for this part of the assignment. Verify the contents of the p4B directory as shown below:

```
[skrentny@jimbo] (45)$ ls
p4B/ p4Bfiles.tgz
[skrentny@jimbo] (46)$ cd p4B
```

```
[skrentny@jimbo] (47)$ ls
CPPLINT.cfg  cpplint.py*  csim.c  csim-ref*  Makefile  README  test-csim*  traces/
```

CSIM

You'll be coding a basic cache simulator in the provided source file, "**csim.c**". This simulator takes a memory trace as input, simulates the hit/miss/eviction behavior of cache memory on the trace, and then **outputs the total number of hits, misses, and evictions**.

Memory Trace Files:

The "**p4B/traces**" directory contains a collection of memory trace files that we will use to evaluate the correctness of your csim implementation. You do not need to generate any traces on your own. However, these memory 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 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:

```
 operation address,size
```

The first character on a line is a space (not visible above). Next is the "operation" that denotes the type of memory access:

- **I**: instruction fetch/read
- **L**: a data load/read
- **S**: a data store/write
- **M**: a data modify (i.e., a data load followed by a data store)

Your simulator will consider only memory accesses to data (L/S/M). It ignores instruction fetches (I). A space is between the operation and the "address", which specifies a **64-bit** hexadecimal memory address. We've been dealing with 32-bit addresses but, **for part B of the assignment, all addresses are 64 bits in length (8 bytes).** The address is followed by a comma and finally the "size", which specifies the number of bytes the operation accesses starting at the specified address.

Using csim:

We have provided you with skeleton code in file "**csim.c**". Your job is to complete the implementation so that it outputs the correct number of hits, misses, and evictions so that it matches the reference cache simulator we've provided, named "**csim-ref**". This executable simulates the behavior of a cache with arbitrary size and associativity on a memory 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:

```
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 cache 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
```

For example, the following command gives us the output:

```
[skrentny@jimbo] (50)$ ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

The same example in verbose mode (assume addresses shown are in hexadecimal):

```
[skrentny@jimbo] (51)$ ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace -v
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
```

The additional output above for verbose mode indicates the information read from the trace (i.e., operation address,size as explained above) and whether that led to hit/miss/eviction in the cache. Use the verbose mode from "**csim-ref**" to compare against your code ("**csim.c**") while debugging. It is highly recommended, **but it is not required**, that you implement the verbose mode in your implementation to help you with debugging.

Compiling and Running csim:

After you've made edits to the source file, "**csim.c**", compile it using the Makefile we've provided and run it as shown below:

```
[skrentny@jimbo] (55)$ make clean
[skrentny@jimbo] (56)$ make
[skrentny@jimbo] (57)$ ./csim -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

Do not print anything else. Only verbose mode can have additional output.

However, you must print error messages when checking for the return value of library functions. You can print any error message that suitably describes the error. Notice that the output from "**csim**" matches the output from "**csim-ref**". To work correctly, your simulator output must match the output of "**csim-ref**" for arbitrary traces and values of **s**, **E**, and **b**.

Comparing csim and csim-ref:

Use the program we've provided, named "**test-csim**", to check your implementation in "**csim.c**" against "**csim-ref**". Run the following command:

```
[skrentny@jimbo] (66)$ ./test-csim
```

The output will show you how your implementation in "**csim.c**" compares against "**csim-ref**". The maximum score possible using the "**test-csim**" program is 48.

SPECIFICATIONS

Most of the specifications are found in the comments of the skeleton source file, "**csim.c**".

Make sure you understand the various variable types that are provided to you in the skeleton code namely **`cache_t`**, **`cache_set_t`**, and **`cache_line_t`**. You will need to decide how to implement the LRU policy for your cache. Two possible implementations are:

- **List:** Move the most recently used cache line to the head of the list. Thus, each set will point to the most recently used cache line. Evict the tail of the list as the least recently used.
- **Counter:** Every cache line has a counter. An access to a line would set its counter value to 1 greater than the current maximum value of all counters in its set. The line with the smallest counter value is the least recently used and is evicted.

Based on the choice you make, you would need to add an extra field to the **`cache_line_t`** struct.

Starting from **`main()`** function, the flow is described below in brief:

- Parse the command line arguments. This is already done for you.
- **`void init_cache()`**:This function should allocate the data structures to hold information about the sets and cache lines using **`malloc()`** depending on the values of parameters S ($S = 2^s$) and E. You need to complete this function.
- **`void replay_trace(char* trace_fn)`**: This function parses the input trace file. This part is already done for you. It should call the **`access_data()`** function. The number of times you call

the **`access_data()`** function depends on the type of instruction it is, namely, if it is a load/store or a modify. Remember, a modify is a data load followed by a data store. You need to complete the missing code in this function.

- **`void access_data(mem_addr_t addr)`**: This function is the core of implementation which should use the data structures that were allocated in **`init_cache()`** function to model the cache hits, misses and evictions. You need to complete this function. The most crucial thing is to update the global variables **`hit_count`**, **`miss_count`**, **`eviction_count`** inside this function appropriately. You should implement the **Least-Recently-Used (LRU) cache replacement policy**.
- **`void free_cache()`**: This function should free up any memory you allocated using **`malloc()`** in **`init_cache()`** function. This is crucial to avoid memory leaks in the code. You need to complete this function.
- **`print_summary(hit_count, miss_count, eviction_count)`**: This function prints the statistics in the desired format. This is already implemented for you.

REQUIREMENTS

- Your program must follow these **Style Guidelines**.
- **NEW: For grading on style, we are going to use Google's lint program** to perform basic style checking. The lint program and it's configuration file can be found in your p4B directory. To obtain full points for style, the following command should not produce any errors: `cpplint.py --extensions=c,h csim.c` Tools like this are commonly used in industry so it is beneficial to have some experience using them.
- Your program must follow these **Commenting Guidelines**. Keep the file header and function header comments we've put in the skeleton code, but you must comment the rest of your code inside the functions where necessary.
- Your programs must operate exactly as specified.
- Your program must check return values of library functions that use return values to indicate errors, e.g., malloc(). Handle errors by displaying an appropriate error message and then calling exit(1).
- We'll compile your programs on the CS Linux lab machine using the Makefile in the p4B directory. It is your responsibility to ensure that your program compiles on these machines, without warnings or errors.

SUBMITTING Your Work

Submit the following **source file** under Project p4B in Assignments on Canvas before the deadline:

1. csim.c

It is your responsibility to ensure your submission is complete with the correct file names having the correct contents. The following points will seem obvious to most, but we've found we must explicitly state them otherwise some students will request special treatment for their carelessness:

- **You will only receive credit for the files that you submit.** You will not receive credit for files that you do not submit. Forgetting to submit, not submitting all the listed files, or submitting executable files or other wrong files will result in you losing credit for the assignment.
- **Do not zip, compress or submit your files in a folder, or submit each file individually.** Submit only the text files as listed as a single submission.
- **Make sure your file names exactly match those listed.** If you resubmit your work, Canvas will modify the file names by appending a hyphen and a number (e.g., csim-1.c) and these Canvas modified names are accepted for grading.

Repeated Submission: You may resubmit your work repeatedly until the deadline has passed. **We strongly encourage you to use Canvas as a place to store a backup copy of your work.** If you resubmit, you must resubmit all of your work rather than updating just some of the files.

Project p4 (1)

Criteria	Ratings		Pts
csim compiles without warnings or errors	**5.0 pts Full Marks**	**0.0 pts No Marks**	5.0 pts
Follows the style and commenting guide As stated in the p4b requirements, we used the cpplint tool to evaluate this criterion. 1 point was deducted for each warning the tool gave. A zero indicates you had more than 5 warnings.	**5.0 pts Full Marks**	**0.0 pts No Marks**	5.0 pts
Checks return value of malloc	**5.0 pts Full Marks**	**0.0 pts No Marks**	5.0 pts
Frees all dynamically allocated memory	**5.0 pts Full Marks**	**0.0 pts No Marks**	5.0 pts
(1,1,1) yi2.trace	**3.0 pts Full Marks**	**0.0 pts No Marks**	3.0 pts
(4,2,4) yi.trace	**6.0 pts Full Marks**	**0.0 pts No Marks**	6.0 pts
(2,1,4) dave.trace	**6.0 pts Full Marks**	**0.0 pts No Marks**	6.0 pts
(2,1,3) trans.trace	**6.0 pts Full Marks**	**0.0 pts No Marks**	6.0 pts
(2,2,3) trans.trace	**6.0 pts Full Marks**	**0.0 pts No Marks**	6.0 pts

Criteria	Ratings		Pts
(2,4,3) trans.trace	**6.0 pts** **Full Marks**	**0.0 pts** **No Marks**	6.0 pts
(5,1,5) trans.trace	**6.0 pts** **Full Marks**	**0.0 pts** **No Marks**	6.0 pts
(5,1,5) long.trace	**9.0 pts** **Full Marks**	**0.0 pts** **No Marks**	9.0 pts
(5,4,5) long.trace	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
(12,1,2) cache1D.trace	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
(8,1,6) cache2Drows.trace	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
(8,1,6) cache2Dcols.trace	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
			Total Points: 80.0