p4B: Coding a Cache Simulator

Due Nov 11 by 11:59pm **Points** 80 **Submitting** a file upload
Available until Nov 13 at 11:59pm

This assignment was locked Nov 13 at 11:59pm.

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

First, you'll need to understand the project code that we've provided. Copy the entire contents from the following directory into your working directory for this assignment:

```
/p/course/cs354-deppeler/public/code/p4B
```

Make sure your directory layout is the same as we've provided, and your copy has a subdirectory named "traces".

```
[deppeler@liederkranz] (44)$ ls
csim.c  csim-ref*  Makefile  test-csim*  traces/
[deppeler@liederkranz] (45)$ cd traces
[deppeler@liederkranz] (46)$ ls
debug  trace1  trace2  trace3  trace4  trace5
```

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 s bits for set index
-E <E>: number of lines per set (associativity)
-b <b>: Number of b bits for block offsets
-t <tracefile>: Name of the valgrind trace to replay
```

For example, the following command gives us the output:

```
[deppeler@liederkranz] (50)$ ./csim-ref -s 4 -E 1 -b 4 -t traces/trace2
hits:4 misses:5 evictions:3
```

The same example in verbose mode (**addresses shown below are treated as hexadecimal**):

```
[deppeler@liederkranz] (51)$ ./csim-ref -s 4 -E 1 -b 4 -t traces/trace2 -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:

```
[deppeler@liederkranz] (55)$ make clean
[deppeler@liederkranz] (56)$ make
[deppeler@liederkranz] (57)$ ./csim -s 4 -E 1 -b 4 -t traces/trace2
hits:4 misses:5 evictions:3
```

Do not print anything else unless it is an error message. Additional output should only appear in verbose mode.

As usual, 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**" shown above matches the output from "**csim-ref**" when called with the same command line arguments. To work correctly, your simulator output must match the output of "**csim-ref**" for arbitrary traces and values

of **s**, **E**, and **b**. You may assume the block size as determined by **b** will be greater than or equal to that maximum size of the bytes accessed by operations in a trace.

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:

```
[deppeler@liederkranz] (66)$ ./test-csim
```

The output will show you how your implementation in "**csim.c**" compares against "**csim-ref**".

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.

In the order of the execution flow from **`main()`**, the following lists work to be completed:

- **DONE:** Parse the command line arguments.
- **COMPLETE this function:** `void init_cache()`
  This function should allocate the data structures using `malloc()` to hold information about the sets and cache lines depending on the values of parameters S ($S = 2^s$) and E.
- **COMPLETE missing code in this function:** `void replay_trace(char* trace_fn)`
  This function parses the input trace file, which is already done for you. It should call the `access_data()` function. The number of times you call the `access_data()` function depends on if a load, store or modify is done. Remember, a modify is a data load followed by a data store.
- **COMPLETE 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 and simulates and tracks the cache hits, misses and evictions. The most crucial thing is to update the global variables `hit_cnt`, `miss_cnt`, `evict_cnt` inside this function appropriately. You should implement the **Least-Recently-Used (LRU) cache replacement policy**.

- **COMPLETE this function:** `void free_cache()`
  This function should free up memory you allocated using `malloc()` in the `init_cache()` function. This is crucial to avoid memory leaks in the code.
- **DONE:** `print_summary(hit_cnt, miss_cnt, evict_cnt)`
  This function prints the statistics in the desired format.

REQUIREMENTS

- Your program must follow style guidelines as given in the **Style Guide**.
- Your program must follow commenting guidelines as given in the **Commenting Guide**. Keep the function header comments we've put in the skeleton code.
- 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

SUBMISSION HAS BEEN ENABLED.

Leave plenty of time before the deadline to complete the two steps for submission found below. There is a 24 hour grace period after the deadline for emergencies. Submitting during this grace period results in your submission being marked late but it will be accepted for grading without penalty. No submissions or updates to submissions are accepted after this grace period.

1.) Submit only the file listed below under Project p4B in Assignments on Canvas. Do not zip, compress, or submit your file in a folder.

- **csim.c**

Repeated Submission: You may resubmit your work repeatedly so we strongly encourage you to use Canvas to store a backup of your current work. If you resubmit, Canvas will modify your file names by appending a hyphen and a number (e.g., csim-1.c).

2.) Verify your submission to ensure it is complete and correct. If not, resubmit all of your work rather than updating just some of the files.

- **Make sure you have submitted all the files listed above.** Forgetting to submit or not submitting one or more of the listed files will result in you losing credit for the assignment.
- **Make sure the files that you have submitted have the correct contents.** Submitting the wrong version of your files, empty files, skeleton files, executable files, corrupted files, or other wrong files will result in you losing credit for the assignment.

- **Make sure your file names exactly match those listed above.** If you resubmit your work, Canvas will modify your file names as mentioned in **Repeated Submission** above. These Canvas modified names are accepted for grading.

Project p4B

Criteria	Ratings		Pts
1. Code compiles without warnings or errors	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
2. Follow the style and commenting guide	**5.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
3. Check the return value of malloc() and calloc()	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
4. Free all dynamically allocated memory	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Execution test: (1,1,1) trace1	**6.0 pts** **Full Marks**	**0.0 pts** **No Marks**	6.0 pts
Execution test: (4,2,4) trace2	**6.0 pts** **Full Marks**	**0.0 pts** **No Marks**	6.0 pts
Execution test: (2,1,4) trace3	**6.0 pts** **Full Marks**	**0.0 pts** **No Marks**	6.0 pts
Execution test: (2,1,3) trace4	**6.0 pts** **Full Marks**	**0.0 pts** **No Marks**	6.0 pts
Execution test: (2,2,3) trace4	**6.0 pts** **Full Marks**	**0.0 pts** **No Marks**	6.0 pts
Execution test: (2,4,3) trace4	**6.0 pts** **Full Marks**	**0.0 pts** **No Marks**	6.0 pts
Execution test: (5,1,5) trace4	**6.0 pts** **Full Marks**	**0.0 pts** **No Marks**	6.0 pts

Criteria	Ratings		Pts
Execution test: (5,1,5) trace5	**9.0 pts** **Full Marks**	**0.0 pts** **No Marks**	9.0 pts
Execution test: (12,1,2) cache1D	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
Execution test: (8,1,6) cache2Drows	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
Execution test: (8,1,6) cache2Dcols	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
			Total Points: 80.0