Computer Systems Principles

Cache Simulator

Contents

Cache Simulator

Overview

This assignment will help you understand the impact that cache memories have on performance of your C programs. It requires you to implement several functions within the context of a program that simulates the behavior of a cache memory. The functions you need to implement focus on different areas that we have most recently studied. In particular:

- **Allocation and Linked Lists**: You are to implement functions that require you to allocate the proper data structures to implement the cache simulator. You will also be required to manipulate a linked list. Knowledge from previous assignments will help here!
- **Bit Extraction**: You are to implement functions that extract bits from an address to index into the cache correctly to determine if you have a *hit* or a *miss*, etc.
- **LRU**: You are to implement the core of the *least recently used* algorithm to determine which cache line to evict when a particular set is full.

Source Files and Compilation

You should begin by downloading the startup tar ball and copying it to your Virtual Box environment. After you copy the archive file to your Virtual Box environment, you should execute the following command:

```
$ tar xzvf cache-proj-student.tgz
```

This will create a directory called `cache-proj` that contains anumber of files and directories that you will work with. The three files that you will be modifying are `src/cache.c`, `src/bits.c`, and `src/lru.c`. Here is a brief summary of the files included in this assignment:

Table 1: Source Files

Files	Description
`Makefile`	This file builds the source code.
`cache-sim`	This is your built executable file.
`cache-sims`	This is the solution executable file.
`src/bits.h,src/bits.c`	The bits interface/implementation
`src/cache.h,src/cache.cc`	The cache interface/implementation.
`src/cpu.h,src/cpu.c`	The cpu interface/implementation.
`src/lru.h,src/lru.c`	The lru intreface/implementation.
`src/main.c`	The application entry point.
`src/test-sim.c`	A test application to test your code.
`src/trace.h`	An interface used by several files.
`src/soln-bits.o`	Object file used by test-sim.
`trace/cat.trace`	A trace file of the cat command.
`trace/ls.trace`	A trace file of the ls command.
`trace/wc.trace`	A trace file of the wc command.

To compile these files, type:

```
$ make clean
$ make
```

`make clean` will delete all of the generated files and `make` will compile all of the source files and generate two binary executable files: `cache-sim` and `test-sim`. The `cache-sim` binary is your cache simulator and the `test-sim` binary is a test harness that will test various aspects of your code to determine the correctness. You are provided a framework that compiles from the beginning. Although it compiles, it does not work yet—you must provide the implementation. **You must use the Virtual Box environment to compile and run the binary files**. It is important that you compile your code in the Virtual Box environment because the representation of the trace addresses are assumed to be 32-bit, etc.

We have provided to you the solution binary so that you can see what the output is of running the cache simulator. To try it out you can run the following:

```
$ ./cache-sims 4 2 6 traces/ls.trace
hits: 2296281 misses: 142489 evictions: 142457 hrate: 0.941573 mrate: 0.058427
```

The `cache-sims` program takes *four* arguments (identical to `cache-sim`). If you run the solution without the arguments you will see a usage print out:

```
$ ./cache-sims
usage: cache-sim <set bits> <associativity> <block bits> <tracefile>
```

You must provide the *number of bits* used to identify the set, the associativity of the cache (number of lines per set), the *number of bits* to identify the byte within a block, and the trace file to run the cache simulator on.

Description

The assignment requires you to complete the implementation of a cache simulator. You can test your cache simulator on trace files that we provide. We will first describe the trace files and then explain the parts that you must implement.

Reference Trace Files

The assignment requires you to implement a cache simulator that will read in an *address trace* of running an actual program. We have generated these trace files using the `valgrind` tool. You can run this tool in the Virtual Box environment to generate additional trace files if you would like to test your simulator on address traces generated from other programs. For example, to generate a trace of the `ls` command, type the following:

```
$ valgrind --log-file=ls.trace --tool=lackey --trace-mem=yes ls -l
```

This will capture the memory accesses of the program in the order they occur and save them to the file `ls.trace`. These `valgrind` trace files have the following format:

```
I  04004c85,1
 S bec6025c,4
I  04004c86,3
I  04004c89,5
 S bec601fc,4
I  04018bab,3
 L bec601fc,4
I  04018bae,1
 L bec601fc,4
```

Each line denotes a memory access. The format of each line is:

`[space]operation address,size`

The *operation* indicates the type of memory access: `I` indicates an instruction fetch, `S` indicates a store, and `L` indicates a load. The *address* is the hexadecimal representation of the address, and the *size* is the number of bytes accessed by the operation.

Cache Simulator Implementation

The objective of this assignment is four-fold:

- To understand a larger C program that includes multiple header and implementation files.
- To understand how C data structures are implemented, allocated, and freed.
- To understand the structure of an address as viewed by a cache and how to use C bit-level operations to extract the right values to index the cache properly.
- To understand and implement the *least-recently-used* (LRU) algorithm to identify which cache lines to evict from the cache.

The *three* files that you must modify are `src/cache.c`, `src/bits.c`, and `src/lru.c`. You should take some time to understand the structure of the code by reading through all the source files. You should also note that we have labeled the places in the code with `TODO` to indicate where you need to provide an implementation. There are *nine* places in the code marked as `TODO` that you must implement. We will break each part down as a task to provide guidance as to how you should approach this assignment.

Task 1: Your first task should be to properly allocate the cache data structure. You should look at `cache.h` to see the different C structures that have been defined. In `cache.c` you will need to implement `make_cache`, `make_sets`, `make_lines`, and `make_block` to properly allocate the cache data structure. The `make_cache` function is called in `main.c` to create a `Cache` object. This is used by the rest of the simulator. You should compile (`make`) your simulator often to ensure that it compiles. You may also periodically run the `test-sim` binary to see if you are able to pass the tests. There are some comments in the code to provide some guidance as to how this should be implemented.

Note: You will need to use `malloc` to allocate memory for each of the necessary data structures. If you are unsure of the structure you should explore how they are used in other parts of the code. You should also take note that the data structures that make up the *cache* are **not linked lists**. They are *arrays* that you must allocate using `malloc` given the proper size. You should use the `sizeof` operator to get the size of the structures to determine how much memory you will need to allocate.

Task 2: Next, you should implement the `get_set`, `get_line`, and `get_byte` functions to extract the proper values from an address. You can find these functions in `bits.c`. You should follow the structure of an address as outlined in class as well as in the book

(page 596, Figure 6.27 (b)). You will need to use your bit-wise operators (e.g., &, |, <<, >>) to extract the proper values.

Task 3: Lastly, you are to implement the LRU algorithm to properly evict the least recently used lines from the cache. We have provided the code to create the LRU data structure. The function you must implement is `lru_fetch`. This function takes *three* arguments: a `Set *`, a `Tag *`, and an `LRUResult *`. The first two are to be used by the `lru_fetch` function to determine an available line and the third is to be used to return the *line* and whether or not the result was a `HIT`, `COLD_MISS`, or `CONFLICT_MISS`—you can find these definitions in `cache.h`. The general approach you should take is to get the `lru_queue` from the `set` and then iterate over this linked list to find the correct line. Here is some pseudocode for the `lru_fetch` function that might help with this:

```
for (LRUNODE **prevp = &(set->lru_queue);
     (*prevp) != NULL;
     prevp = &((*prevp)->next)) {
  current = *prevp
  line = current->line
  if the line is valid and the tag matches:
    we have a hit
    set the result access to HIT


  else if the line is not valid:
    we did not find a matching tag so the set is not full
      set the line valid to 1
      set the line tag to the tag
      set the result access to COLD_MISS

  else if the line is the last line
    we did not find a matching tag and the set is full
      set the line tag to the tag
      set the result access to CONFLICT_MISS

  else
    continue to the next element

  unchain current from the list
  place current at the front
  set the result line to the line
  return (Note: this is intentionally inside the for)
```

If you implement everything correctly your `test-sim` program will output the following:

```
=====================================================
Test 01 PASSED: cache is not NULL
```

```
Test 02 PASSED: cache->set_bits properly initialized
Test 03 PASSED: cache->block_bits properly initialized
Test 04 PASSED: cache->line_count properly initialized
Test 05 PASSED: cache->set_count properly initialized
Test 06 PASSED: cache->block_count properly initialized
Test 07 PASSED: cache->sets is not NULL
Test 08 PASSED: line_count in Set is properly initialized
Test 09 PASSED: Your LRU queues have been properly initialized
Test 10 PASSED: Your LRU queues appear to have the correct number of nodes (4)
Test 11 PASSED: lines in Set is not NULL
Test 12 PASSED: blocks and block_count in Line is properly initialized
=====================================================
Test 13 PASSED: get_set is returning the proper set value
Test 14 PASSED: get_line is returning the proper tag value
Test 15 PASSED: get_block is returning the proper block index
=====================================================
hits: 803213 misses: 1354 evictions: 1290 hrate: 0.998317 mrate: 0.001683
Test 16 PASSED: the number of misses (cpu->cold+cpu->conflict) is correct (1354)
Test 17 PASSED: the number of hits (cpu->hits) is correct (803213)
Test 18 PASSED: the number of evictions (cpu->conflict) is correct (1290)
hits: 2016836 misses: 421934 evictions: 421918 hrate: 0.826989 mrate: 0.173011
Test 19 PASSED: the number of misses (cpu->cold+cpu->conflict) is correct (421934)
Test 20 PASSED: the number of hits (cpu->hits) is correct (2016836)
Test 21 PASSED: the number of evictions (cpu->conflict) is correct (421918)
hits: 787666 misses: 22104 evictions: 22088 hrate: 0.972703 mrate: 0.027297
Test 22 PASSED: the number of misses (cpu->cold+cpu->conflict) is correct (22104)
Test 23 PASSED: the number of hits (cpu->hits) is correct (787666)
Test 24 PASSED: the number of evictions (cpu->conflict) is correct (22088)
=====================================================
You passed 24 out of 24 tests.
You scored 70/70 points.
Nice Work!
=====================================================
```

Evaluation

You will be evaluated based on the number of tests you pass (70 points) and the usual peer assessment categories. The number-of-tests result has a relative weight of 72 while each of the 5 other criteria have relative weight of 6.

It is possible that a faulty submission will cause `test-sim` not to output the total score of tests passed. To help you determine the score in such cases, here is the point value of each test:

Test	Points
01	0
02	4
03	4
04	2
05	3
06	3
07	3
08	1
09	1
10	3
11	2
12	2
13	2
14	2
15	2
16	4
17	4
18	4
19	4
20	4
21	4
22	4
23	4
24	4

Submission

You must submit your assignment as a *tarball*. After you complete the assignment you need to run the following command from the *parent* directory of your `cache-proj-student` directory. So, assuming you are already in your `cache-proj-student` directory you want to execute the following commands from the command line:

```
$ make tar
```

This will create the file `cache-proj-submit.tgz` which you need to upload to the cache simulation assignment activity in Moodle. **Make sure you upload this file and not the original tarball!** Please submit your assignment to Moodle by the assigned due date. Please make sure you have followed all the instructions described in this assignment. Failure to follow these instructions precisely will likely lead to considerable point deductions and possibly failure for the assignment. We remind you also of our academic honesty policy and that of the campus.