CSCI 2122 Assignment 5

Due date: 11:59pm, Tuesday, April 9, 2024, submitted via git

Objectives

The purpose of this assignment is to practice your coding in C, and to reinforce the concepts discussed in class on pointers, caching, and the memory hierarchy.

In this assignment you will implement a cache simulator that uses a limited amount of memory.

Preparation:

1. Complete Assignment 0 or ensure that the tools you would need to complete it are installed.
2. Clone your assignment repository:

   ```
   https://git.cs.dal.ca/courses/2024-winter/csci-2122/assignment-5/????.git
   ```

   where ???? is your CSID. Please see instructions in Assignment 0 and the tutorials on Brightspace if you are not sure how.

Inside the repository there is one directory: **`cachex`**, where code is to be written. You should set up a CLion project for this directory. Inside the directory is a **`tests`** directory that contains tests that will be executed each time you submit your code. Please do not modify the **`tests`** directory or the **`.gitlab-ci.yml`** file that is found in the root directory. Modifying these files may break the tests. These files will be replaced with originals when the assignments are graded. You are provided with a sample **`Makefile`** file that can be used to build your program. If you are using CLion, a **`Makefile`** will be generated from the **`CMakeLists.txt`** file generated by CLion.

Background:

Fast memory is expensive. Cache designers are limited by the amount of fast memory that they can use. Furthermore, the fast memory must store not only the data being cached but all the metadata as well, such as tags, the valid bit, and the timestamp. Naturally, cache designers simulate their designs in software before implementing in hardware. In this assignment, you will do the same.

Your task is to implement a cache module that simulates a cache. The choice of the type of cache is up to you. Your cache module will be provided with two parameters: ***F***, the amount of "fast" memory that your cache may use and ***M***, the amount of memory in the simulated system, from which data will be cached. As well, your module will be provided a pointer to the "fast" memory of size ***F***. Your cache module may ONLY use this "fast" memory (apart from local variables) to implement the cache. In short, any data that is needed to manage the cache, as well as the data being cached, must be stored in the "fast" memory.

Caches

Recall that a cache is defined by several parameters:

S: the number of sets
E: the number of lines per set
B: the number of bytes each line caches

The size of the cache is C = S x E x B. The type of cache depends on these parameters:

- In **direct mapped caches,** E = 1, i.e., there is one (1) line per set,
- In **fully associative caches**, S = 1, i.e., all lines are in a single set, and
- In **set associative caches**, S > 1 and E > 1, i.e., there are multiple sets, each with multiple lines.

When a cache receives a memory reference, it

1. Breaks up the address into a **tag**, a set **index**, and an **offset**.
2. Uses the **index** to identify the set where the referenced memory may be cached
3. Uses the **tag** to determine if a line in the set is caching the referenced memory
4. If the referenced memory is not being cached,
    a. The cache determines if the set contains an unused line
    b. If no lines are unused, the cache will evict a used line.
    c. The line is then loaded with the block of memory containing the reference.
5. At this point, a line in the selected set is caching the memory being referenced. The cache returns the data being referenced.

The choice of what type of cache to use, is completely up to you. The only restriction is that all parts of the cache must fit into ***F*** bytes of the "fast" memory and that the line size *B* ≥ 32 bytes.

Reference Streams

A reference stream is simply a sequence of memory references (addresses), representing a program running and accessing memory as it runs. The first integer, ***R***, denotes the number of memory references that follow. The next ***R*** integers are the memory addresses.

```
9
22
48
70
4118
22
4118
2070
4118
22
```

The Cache Simulator

Your cache simulator takes as input (i) a system configuration that includes:

- ***F_size***: Size of the "fast" memory where ***F_size*** ≥ 256
- ***F_memory***: A pointer to the "fast" memory
- ***M_size***: Size of main memory

(ii) a reference stream, and (iii) an optional "stats" command that causes the simulator to print out the cache hit/miss count and hit-rate. The simulator instantiates the system being simulated and then processes the reference stream, by sending each reference to the cache. The cache will forward the request to the main memory, if the request causes a miss, loading a line from the memory. Once the requested data is in the cache, the cache returns the requested data. The simulator counts the hits and misses that occur, and can output the hit-rate after the reference stream completes.

Your task will be to implement the cache module in the Cache Simulator.

Task: Implement the `cache.c` for the Simulator

Your task is to implement the `cache.c` module by implementing one function. The function is declared in `cache.h` and is called from `main.c`. The function is:

```
int cache_get(unsigned long address, unsigned long *value)
```

This function takes a memory **address** and a pointer to a **value** and loads a word (a long) located at memory **address** and copies it into the location pointed to by **value**. I.e., this is what the CPU does when it needs to load a word from memory, it requests it from the cache. The function takes two (2) parameters:

- **address** : the location of the value to be loaded. Addresses are the memory references from the reference stream.
- **value** : a pointer to a buffer of where the word is to be copied into.

The function returns a 1 on success and a 0 on failure. The function performs two steps:

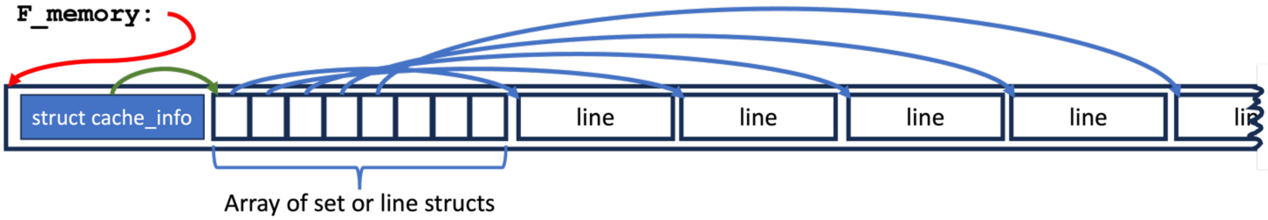
1. Check if the cache system has been initialized. If not, the initialize the cache.
2. Process the request, by returning the value at the specified memory address.

Step 1: Checking and Initializing the Cache

The function has access to a global struct called ***c_info***, which is defined in `cache.h`. The struct is

```
    struct cache_info {
        void *F_memory;          /* pointer to "fast" memory */
        unsigned int F_size;     /* size of "fast" memory (in bytes) */
        unsigned int M_size;     /* size of main memory (in bytes) */
    };
```

The pointer **`c_info.F_memory`** points to a memory chunk of size **`c_info.F_size`**. The memory is initialized to all 0s. This is the **only** memory, except for local variables that you may use in implementing the cache. You may **not** use `calloc()` or `malloc()`, or create any additional static or global variables. The recommended approach is to define a struct and place it at the start of the "fast" memory pointed to by **`F_memory`**. The struct can point to an array of structs representing sets or lines also located in the "fast" memory. These structs can contain pointers, pointing to lines (that store data), and which are also kept in the "fast" memory.



Have an "initialized" flag in the struct at the start of the "fast" memory that is set to 1 if the cache is initialized and 0 otherwise. **Hint:** create a static **`init()`** function in `cache.c` that is called from `cache_get()` if the "initialized" flag is 0. The **`init()`** function can then set up all the pointers and structures. Note: It is up to you to decide on how many sets and lines the cache will have. The only restrictions are (1) The **minimum** size of a line (B) must be 32 bytes. And (ii) everything must fit into **`F_size`** bytes of memory. **`F_size`** will be greater or equal to 256. **Reminder**: one of the things that **`init()`** should do is set the initialized flag to 1.

Step 2: Processing a Request

To process a request, the `cache_get()` function should:

1. Break up the address into a *tag*, *index*, and *offset*.
2. Use the index to locate the correct set.
3. Use the tag to determine if block of memory that includes the **address** is in one of the lines in the set.
4. If it is (a cache hit), the offset is used to locate the word in that line, the word should be copied into the buffer pointed to by **value**, and then the function returns.
5. Otherwise, it is a cache miss. In this case, a victim line is selected, initialized with the tag of the needed memory block, and loaded by calling the function

   ```
   int memget(unsigned int address, void *buffer, unsigned int size)
   ```

   which is declared in `cache.h` and defined in `main.c`. This function takes the **address** as the first parameter, a pointer to a **buffer** where the block should be loaded, and the **size** of the block to get. Hint, the buffer should point to the part of the line storing the block. The function returns 1 on success and 0 on failure. Each call to `memget()` counts as a miss.

The rest of the cache simulator is already implemented for you! ☺

The cachex Mainline

The `main.c` of `cachex` is already implemented for you. Below is a brief description of what it does.

Input

The cachex reads input from stdin. The input consists of three parts: (i) a system configuration; (ii) a reference stream; and (iii) an optional "**`stats`**" command.

The system configuration consists of two integers:

- ***F*** : the "fast" memory size
- ***M*** : the memory size

The reference stream consists of an integer ***N*** denoting the number of references, followed by ***N*** references. Each reference is an integer between 0 and ***M*** – 8, denoting the address in memory being referenced.

After the ***N*** memory references, and optional "**`stats`**" command may be present. This command consists of a single word "**`stats`**" and causes the simulator to print out the hits, misses, and hit-rate.

Processing

When `cachex` starts running, it reads in the system configuration, allocates the memory in the system being simulated, and initializes the **`c_info`** struct. The main memory is initialized to a sequence of pseudorandom numbers (the numbers look random, but they are not).

It then enters the main loop and processes the reference stream:

- For each reference, `cache_get()` is called.
- The loaded value is compared to the expected value and any errors are noted.

During the processing, all cache hits and misses are recorded.

If the "**`stats`**" command is present after the memory references, the number of hits and misses is displayed.

Output

The `cachex` outputs to stdout in two parts: (i) the result of each memory reference as it is being processed; (iii) the aggregates of hits and misses, if the stats command was used.

Example

Input	Output
`1024 65536`	`Loaded value [0xb9cb17b29e5109d2] @ address 0x00000016`
`9`	`Loaded value [0x0394fee63984c8dc] @ address 0x00000030`
`22`	`Loaded value [0x8eba29a6bb1465ff] @ address 0x00000046`
`48`	`Loaded value [0x3ce65cc676176add] @ address 0x00001016`
`70`	`Loaded value [0xb9cb17b29e5109d2] @ address 0x00000016`
`4118`	`Loaded value [0x3ce65cc676176add] @ address 0x00001016`
`22`	`Loaded value [0x425a273223d06058] @ address 0x00000816`
`4118`	`Loaded value [0x3ce65cc676176add] @ address 0x00001016`
`2070`	`Loaded value [0xb9cb17b29e5109d2] @ address 0x00000016`
`4118`	`Cache hits: 4, misses: 5`
`22`	
`stats`	

Hints and Suggestions

- You will need a couple structs, one for cache and one for line. You may also want one for set.
- Fundamentally, a cache is an array of sets, and a set is an array of lines.
- You should only need to modify one file: `cache.c`.
- There is not a lot of code to write (my solution under 100 lines).

Grading

If your program does not compile, it is considered non-functional and of extremely poor quality, meaning you will receive 0 for the solution.

The assignment will be graded based on three criteria:

Functionality: "Does it work according to specifications?". This is determined in an automated fashion by running your program on several inputs and ensuring that the outputs match the expected outputs. The score is determined based on the number of tests that your program passes. So, if your program passes t/T tests, you will receive that proportion of the marks.

Performance: "Does it perform well?". This is determined in an semi-automated fashion by running your program on several inputs and the comparing benchmarks of your cache to that of the solution.

Quality of Solution: "Is it a good solution?" This considers whether the approach and algorithm in your solution is correct. This is determined by visual inspection of the code. It is possible to get a good grade on this part even if you have bugs that cause your code to fail some of the tests.

Code Clarity: "Is it well written?" This considers whether the solution is properly formatted, well documented, and follows coding style guidelines. A single overall mark will be assigned for clarity. Please see the Style Guide in the Assignment section of the course in Brightspace.

The following grading scheme will be used:

Task	100%	80%	60%	40%	20%	0%
Functionality (20 marks)	Equal to the number of tests passed.					
Performance (10 marks)	Hit rate of cache meets or exceeds hit rate of the solution	Hit rate of cache is ≥80% of the solution	Hit rate of cache is ≥60% of the solution	Hit rate of cache is ≥40% of the solution	Hit rate of cache is ≥20% of the solution	No code submitted or code does not compile
Solution Quality (10 marks)	Implemented efficiently and correctly.	Implementation is correct. Cache is fully functional.	Minor flaws with implementation. Cache is mostly functional.	Major flaws in implementation. Cache approach looks workable.	An attempt has been made.	
Code Clarity (10 marks) Indentation, formatting, naming, comments	Code looks professional and follows all style guidelines	Code looks good and mostly follows style guidelines.	Code is mostly readable and mostly follows some of the style guidelines	Code is hard to read and follows few of the style guidelines	Code is not legible	

Assignment Submission

Submission and testing are done using Git, Gitlab, and Gitlab CI/CD. You can submit as many times as you wish, up to the deadline. Every time a submission occurs, functional tests are executed, and you can view the results of the tests. To submit use the same procedure as Assignment 0.

Assignment Testing without Submission

Testing via submission can take some time, especially if the server is loaded. You can run the tests without submitting your code by using the provided `runtests.sh` script. Running the script with no arguments will run all the tests. Running the script with the test number, i.e., 00, 01, 02, 03, … 09, will run that specific test. Please see below for how run the script.

Get your program ready to run

If you are developing directly on the unix server,

1. SSH into the remote server and be sure you are in the **cachex** directory.
2. Be sure the program is compiled by running **make**.

If you are using CLion

1. Run your program on the remote server as described in the CLion tutorials.
2. Open a remote host terminal via **Tools → Open Remote Host Terminal**

If you are using VSCode

1. Run your program on the remote server as described in VSCode tutorials.
2. Click on the Terminal pane in the bottom half of the window or via **Terminal → New Terminal**

Run the test script

3. Run the script in the terminal by using the command:

```
./runtest.sh
```

to run all the tests, or specify the test number to run a specific test, e.g. :

```
./runtest.sh 07
```

Run the benchmark script

3. Run the script in the terminal by using the command:

```
./runbench.sh
```

to run all the tests, or specify the test number to run a specific test, e.g. :

```
./runbench.sh 03
```

You will see the bench run in the terminal window.