Assignment 5: A Caching Simulator
CSE 130: Principles of Computer Systems Design

Optional: Due: June 9, 2023 at 11:59 PM

Goals This assignment provides you with experience implementing and testing *real* caching algorithms. Requesting a resource from disk can be time-consuming, especially if the resource needs to be pre-processed. A cache is a fantastic way to reduce the number of disk reads and redundant work done by storing the results of a previous query to be used again. We encourage you to consider your cache's *design* before you start building the code.

Overview In this assignment, you will be implementing a cache. Your cache must support first-in-first-out (FIFO), least-recently-used (LRU), and clock eviction policies. Your program should continuously take items from `stdin` until `stdin` is closed. After each lookup, your program should print to `stdout` specifying whether the item that was accessed is a `HIT` or `MISS` . If the lookup was a miss, your cache will add the item to its cache and evict an item based on the eviction policy that the user specified. Before your program exits, and after `stdin` is closed, you must include a summary line that specifies the total number of compulsory and capacity misses. Note: the eviction algorithms are fully associative, so there are no conflict misses.

Submission As always, the code that you submit must be in your repository on `git.ucsc.edu`. In particular, your assignment must build your cache implementation, called `cacher` , when we execute the command `make` from the `asg5` directory. Submit a 40-character commit ID hash on Canvas to identify the commit that you want us to grade. We will grade the last hash that you submit and will use the timestamp of your last upload to determine grace days. For example, if you post a commit hash 36 hours after the deadline, we will subtract 2 grace days from your total. When you submit, you should only submit the commit ID you want us to grade.

Assignment Details

Your server must implement the functionality explained below subject to the limitations below.

Functionality

Your new `cacher` should take two command line arguments, `size`—the size of your cache—`policy`—the policy your cache will follow. The policy flag will be present and will be exactly `-F` (First-In First-Out), `-L` (Least Recently Used), or `-C` (Clock). Your `cacher` should use First-In First-Out if no policy is given.

```
./cacher [-N size] <policy>
```

Once initialized, `cacher` must continuously take newline-separated items from `stdin` until `stdin` is closed. For each item received, `cacher` should access the cache to determine if the item exists. Depending on if the access resulted in a `HIT` or `MISS` , `cacher` should update the cache accordingly. Namely, on a `HIT` , the cache should print out "`HIT`\n". On a `MISS` , the cache should print out "`MISS`\n", add the item to the cache, and, if the cache is full, evict an item from the cache following the caching policy. When `stdin` is closed, a summary line should be printed which specifies the number of misses that were compulsory misses and the number that were capacity misses. Note there are no conflict misses since we have a fully associative cache. The following is a table for formatting the required outputs.

	stdout
Cache Hit	"HIT\n"
Cache Miss	"MISS\n"
Summary Line	"<Number of Compulsory Misses> <Number of capacity Misses>''

Additional Functionality

In addition to supporting the methods listed above, `cacher` should not have any memory leaks.

Limitations

You must write `cacher` using the 'C' programming language. It program cannot use functions, like `system` or `execve`, that allow you to execute external programs. **If your submission does not meet these minimum requirements, then the maximum score that you can get is 5%.**

Examples

We include a number of examples to illustrate the intended functionality of `cacher` .

Example 1

This example focuses on cache usage for policy flag `-F` when N=3. The row of the table includes the Item that is accessed next (Item), the state of the cache before the item is accessed (Initial Cache State), the state of the cache after the item is accessed (Updated Cache State), the output of `stdout` , the number of Compulsory misses up after that item is accessed (CO), and the number of Capacity misses after that item is accessed (CA).

No.	Initial Cache State	Item	Updated Cache State	stdout	CO	CA
0.	[ ] [ ] [ ]	a	[a] [ ] [ ]	MISS\n	1	0
1.	[a] [ ] [ ]	b	[a] [b] [ ]	MISS\n	2	0
2.	[a] [b] [ ]	a	[a] [b] [ ]	HIT\n	2	0
3.	[a] [b] [ ]	c	[a] [b] [c]	MISS\n	3	0
4.	[a] [b] [c]	c	[a] [b] [c]	HIT\n	3	0
5.	[a] [b] [c]	d	[b] [c] [d]	MISS\n	4	0
6.	[b] [c] [d]	d	[b] [c] [d]	HIT\n	4	0

Example 2

This example focuses on cache usage for policy flag `-L` when N=3. The row of the table includes the Item that is accessed next (Item), the state of the cache before the item is accessed (Initial Cache State), the state of the cache after the item is accessed (Updated Cache State), the output of `stdout` , the number of Compulsory misses up after that item is accessed (CO), and the number of Capacity misses after that item is accessed (CA). The cache state is ordered by recency-of-access, with the least recent on the left side.

No.	Initial Cache State	Item	Updated Cache State	stdout	CO	CA
0.	[ ] [ ] [ ]	a	[a] [ ] [ ]	MISS\n	1	0
1.	[a] [ ] [ ]	b	[a] [b] [ ]	MISS\n	2	0
2.	[a] [b] [ ]	a	[b] [a] [ ]	HIT\n	2	0
3.	[b] [a] [ ]	c	[b] [a] [c]	MISS\n	3	0
4.	[b] [a] [c]	c	[b] [a] [c]	HIT\n	3	0
5.	[b] [a] [c]	d	[a] [c] [d]	MISS\n	4	0
6.	[a] [c] [d]	c	[a] [d] [c]	HIT\n	4	0

Example 3

This example shows how the cache works with policy flag `-C` when N=3. In contrast to examples 1 and 2, an entry α, β consists of the item α and its referenced bit $\beta = \{$ 0 if unreferenced, 1 if referenced $\}$. Additionally, we include a Next column, indicating the next element to be considered by the clock.

No.	Initial Cache State			Initial Next	Item	Updated Cache State			Updated Next	`stdout`	CO	CA
0.				-	a	a, 0			a	`MISS\n`	1	0
1.	a, 0			a	b	a, 0	b, 0		a	`MISS\n`	2	0
2.	a, 0	b, 0		a	c	a, 0	b, 0	c, 0	a	`MISS\n`	3	0
3.	a, 0	b, 0	c, 0	a	a	a, 1	b, 0	c, 0	a	`HIT\n`	3	0
4.	a, 1	b, 0	c, 0	a	d	a, 0	d, 0	c, 0	c	`MISS\n`	4	0
5.	a, 0	d, 0	c, 0	c	e	a, 0	d, 0	e, 0	a	`MISS\n`	5	0
6.	a, 0	d, 0	e, 0	a	d	a, 0	d, 1	e, 0	a	`HIT\n`	5	0

For No. 4, since the referenced bit for item a was set to 1, and the cache was full, a must be moved to the end and a's referenced bit must be reset prior to b being removed to make room for d.

Rubric

We will use the following rubric for this assignment:

Category	Point Value
Makefile	10
Clang-Format	5
Files	5
Functionality	80
Total	100

Makefile Your repository includes a Makefile with the rules `all` and `cacher`, which produce the `cacher` binary, and the rule `clean`, which removes all `.o` and binary files. Additionally, your Makefile should use clang (i.e., it should set `CC=clang`), and should use the `-Wall`, `-Wextra`, `-Werror`, and `-pedantic` flags (i.e., it should set `CFLAGS=-Wall -Wextra -Werror -pedantic`).

Clang-Format All `.c` and `.h` files in your repository are formatted in accordance with the `.clang-format` file included in your repository.

Files The following files are included in your repository: `cacher.c`, `Makefile`, and `README.md`. Your repository should not include binary files nor any object files (i.e., `.o` files). To make it easier for you to maintain tests, you can also include binary files in any directory whose name starts with the phrase `test`.

Functionality Your `cacher` program performs the functionality described in Assignment Details.

Resources

Here are some resources to help you:

Testing

We provided you with three resources to test your own code:

1. An autograder, which is run each time you push code to GitLab, will show you the points that you will receive for your Makefile, Clang-Format, and Files.

2. A set of test scripts in the resources repository to check your functionality. You can use the tests to see if your functionality is correct by running them on your Ubuntu 22.04 virtual machine. We provided you with a subset of the tests that we will run, but, I bet you can figure the other ones out by adapting what we have given you :-)

Hints

This assignment can be deceivingly simple at first. Refrain from letting this initial assumption get the better of you. Start your project early!

Design Approach

You should design your `cacher` before you start writing any code. Your design should articulate exact function prototypes and struct definitions you plan to use. You may find it helpful to visualize how your `cacher` will function by drawing a picture depicting more extensive procedures shown in the examples section. With a thorough understanding and a clear idea of how to design your `cacher` , you will find that you can avoid many potential bugs, thereby saving you hours! Time is especially crucial for this assignment as you do not have as much as in prior assignments.

Keep in mind that your design will probably change as you build your `cacher` ; sometimes, the act of trying to implement something illuminates how bad of an idea it was. If this happens, we encourage you to "go back to the drawing board" and re-design the `cacher` . If you find yourself implementing something that you haven't "designed", then you're doing it wrong.

Step 1: Design your Scaffolding In any project, it's good to start by thinking about how to get a basic prototype working–something that doesn't actually solve the problems in the assignment but nevertheless has running code. The scaffolding in this project is the cache—How will you create it? What data structure will you use *(hint hint, the cache size N is fixed)*? Pay no attention to handling items from the command line, instead focus on the operations you will need to implement your cache. For example, for `-L` and `-C` , there must be an operation to move any given element to the "end" of your cache. What other operations would you need to implement? Creating a module here may be a good idea.

Step 2: Design your I/O The requirement that you print the result to `stdin` after each result and the summary line should be considered here. If you properly implement step 1, this step should be a matter of printing out data that is already available. A good visual for step 1 and how it makes step 2 easy can be seen through the examples. Notice how the expected output for `stdout` is shown in the `stdout` column and the summary line consists of the last row of the compulsory misses and capacity misses column.

Implementation Approach

Now that you've designed everything, we suggest that you follow those design steps, one by one, when implementing your `cacher` . Start by implementing your scaffolding, then handle your I/O.