1 Introduction

Read the entire document before starting. There are critical pieces of information and hints along the way.

Caches are complex memory and sometimes difficult to understand. One way to understand them is to build them. However, due to time constraints and lack of hardware expertise, we will instead write a cache simulator. In this project, you will be implementing a cache simulator. We have provided you the following files:

- `src/cachesim_driver.c` - Program to runs the simulator and for a trace and calls the methods in `cachesim.c`.
- `src/cachesim.h` - Header file for simulator.
- `src/cachesim.c` - The cache simulator, missing some vital portions of code. *This is the only file you should modify.*
- `Makefile` - Compiles your project.
- `traces/` - Contains the sample Spec Benchmark workloads.
- `solutions/` - The solution generated by the TAs. The `run_script.sh` will produce output that you should compare to `stats.log` for a final confirmation your cache works. Your simulator should produce output that exactly matches this. For debugging along the way, you can use the detailed trace files that say whether each access is a hit or a miss with the default cache configuration for each policy.
- `run_script.sh` - A shell script to run all the test cases.

You will be filling in the functions in the `cachesim.c` file, and then validating your output against the given outputs in the `solutions` folder.

This assignment that will check your understanding of how caches work. **However, keep in mind that your choice of data structure used will greatly impact the difficulty of implementing the cache.** If you are struggling with writing the code, then step back and review the concepts. Be sure to start early, ask Piazza questions, and visit us in office hours for extra help!

2 Cache Specifications

Here are the specifications that your simulator must meet:

1. The simulator must model a cache with 2^C bytes of data storage, with 2^B byte blocks with 2^S blocks per set.
   Note: S = 0 implies a direct mapped cache, and S = C - B is a fully associative cache.
2. The cache implements a **write back, write allocate** policy. Think about what this means in terms of dirty bits.
3. There is a valid bit for each block. This should be set to 0 at the start of the simulation.
4. The cache should be able to implement both an **LRU** replacement and **FIFO** replacement policy. A command line flag will be used to choose which replacement policy your simulation should use.
5. All memory addresses are 64 bits.
6. The cache is byte addressable.

3 Implementation

Your task is to fill out the following functions in `cachesim.c`:

1. `cache_init()`
2. `cache_access()`
3. `cache_cleanup()`
4. `get_tag()`
5. `get_index()`

Each function listed above has helpful comments in the `cachesim.c` file. You may add any global variables or helper functions you deem necessary. You also may need to alter the structs in `cachesim.c` to fit your needs. You should not change anything outside of `cachesim.c` or add any new files. You do not need to and should not use any math libraries to split up addresses in `get_tag()` and `get_index()`.

A trace of memory access (a memory address and whether it is a read or a write) is given in each of the files in the `traces` folder. The driver will first run the `cache_init()` function, which then will run your `cache_access()` with the address for each memory access. Since this is just a simulator, you will not have to worry about any data storage. You will only need to keep track of metadata to simulate the cache accesses. Once the simulator has performed all of the accesses, the simulator will call `cache_cleanup()` and print the statistics.

4 Statistics

The output from your final cache is the statistics that your cache calculates for each workload. Here is the list of fields inside the `cache_stats_t` struct and their meaning:

1. `accesses` - The total number of memory accesses your cache gets
2. `reads` - The number of accesses that were reads
3. `read_misses` - The total number of reads that were cache misses
4. `writes` - The total number of accesses that were writes
5. `write_misses` - The total number of writes that were cache misses
6. `misses` - The total number of cache misses (both reads and writes)
7. `write_backs` - The total number of times data was written back to memory
8. `cache_access_time` - The access time of the cache (3ns)
9. `memory_access_time` - The miss penalty for a cache miss (120ns)
10. `miss_rate` - The miss rate for the cache
11. `avg_access_time` - The average access time for your cache, aka AAT

The driver will format and print the contents of the stats struct to the console. You only need to fill in the stats structure passed in the cache access and cache cleanup functions.

5 Validation Requirement

We have provided sample outputs in the `solutions` directory. You must run your simulator and debug until your output **perfectly** matches all the statistics given in the sample outputs. Use the instructions in section 6.4 to compare your simulator's output against our samples.

6 How to Run / Debug Your Code

6.1 Environment

Your code will need to compile under Ubuntu 16.04 LTS. You can develop on whatever environment you prefer, so long as your code also works in Ubuntu 16.04 (which we will use to grade your projects). **Non-compiling solutions will receive a 0! Make sure your code compiles with no warnings.** We recommend downloading VirtualBox and setting up a new virtual machine with Ubuntu 16.04 if you are on Mac OS or Windows. If you are having trouble with this, come to office hours.

6.2 Compiling and Running

We have provided a Makefile that will run gcc for you. To compile your code with no optimizations (which you should do while developing, it will make debugging easier), just run

```
$ make debug
```

from the main project directory. This will produce the `cachesim` executable, which you can run with

```
$ ./cachesim -C <cache size> -B <block size> -S <set associativity>
        -r <LRU or FIFO> -i <trace name>
```

All arguments are optional. Additionally, the `-p` flag can be used to make your simulator output every memory access and the cache behavior, allowing you to identify individual bugs by comparing to the given solution files for each trace.

6.3 Debugging Tips

You can use valgrind and gdb (GNU Debugger) to help debug your project. Valgrind is a program that helps detect memory management bugs, and GDB is a command line interface that will allow you to set breakpoints, step through your code, see variable values, and identify segfaults. There are tons of online guides, click here (http://condor.depaul.edu/glancast/373class/docs/gdb.html) for one.

To use valgrind, run:

```
$ valgrind ./cachesim -C 15 -B 4 -S 2 -r FIFO -i traces/mcf.trace
```

To start your program in gdb, run:

```
$ gdb ./cachesim
```

Within gdb, you can run your program with the `run` command, see below for an example:

```
(gdb) r -C 15 -B 4 -S 2 -r FIFO -i traces/mcf.trace
```

Feel free to ask about valgrind or gdb and how to use it in office hours and on Piazza. Do not ask a TA or post on Piazza about a segfault without first running your program through both.

6.4 Verifying Your Solution

You can ensure your cache simulator works by testing it and comparing it to the given files in the `solutions` folder. An example of this workflow is given below. You could also use the same tools to check the detailed output traces.

```
$ make clean
$ make
$ ./run_script > my_stats.log
$ diff -u my_stats.log solutions/stats.log
```

The commands above rebuild the project. They then use the `./run_script.sh` script to run your cache with several different configurations. The `>` character redirects the output of the script from `stdout` to a file called `my_stats.log`. Then, the `diff` program will identify the lines that differ between your output and the given solutions in `solutions/stats.log`.

7 Custom Replacement

In order to receive full credit on this assignment, you must design and implement your own replacement policy in addition to FIFO and LRU. Your custom replacement policy must perform with a better AAT than **both** FIFO **and** LRU for **at least one** of the traces.

In the file `README.md`, you must include the following:

- Instructions on how to run your custom replacement policy.
- A description of how your custom algorithm works. There is no length requirement, but you must explain it well enough that another person could understand how to implement it on their own.
- Which trace (or traces) your custom algorithm performs better with, and why you think it may have performed better than FIFO and LRU.

8 How to Submit

Please submit all of the following files in a **.tar.gz** archive. You must turn in:

- Makefile
- README.md
- src/cachesim.c
- src/cachesim.h
- src/cachesim_driver.c

Run `make submit` to automatically package your project for submission.

Always re-download your assignment from T-Square after submitting to ensure that all necessary files were properly uploaded. If what we download does not work, you will get a 0 regardless of what is on your machine.