CS 264 Systems **Fall 2020**
http://comp264.org

Cache Simulator Homework

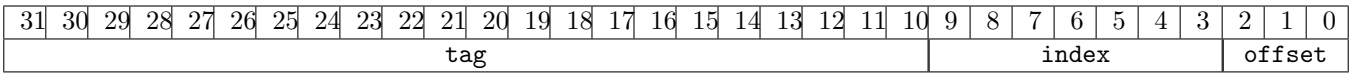
Name:

1 Intro

In this homework, you're going to create a program that simulates the operation of a CPU cache. The learning goal is to (1) understand how a cache operates and (2) strengthen your C programming skills.

2 A Review of the Cache Geometry

We split the address up into three fields: offset, index, and tag. The offset field takes up the lowest `f` bits, the index field is the next `i` bits, and the tag takes the remaining bits. Below I've shown an example where `f = 3` and `i = 7`. The offset in the example is 3 bits long (it takes up bits 0, 1, and 2 in the address), meaning that we have $2^3 = 8$ bytes per cache line. The index is 7 bits long (taking bits 9, 8, 7, 6, 5, 4, 3 in the address), meaning that we have $2^7 = 128$ lines in the cache.



Index	Valid	Tag	Data							
0	1	0x00104F								
1	1	0x00C002								
2	1	0x00C002								
3	1	0x00C002								
4	1	0x00FFF0								
5	1	0x00FFF0								
6	1	0x00104F								
7	1	0x00FF10								
...										
127	1	0x00FF10								

3 Memory Traces

We will be using memory traces generated by `valgrind` to test our cache simulator. `valgrind` is a debugging tool that can be used to track the memory usage patterns of programs as they run. It can also log traces of memory access:

```
pi@raspberrypi ~ $ valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

This command generates a memory trace of the command `ls -l`. See Appendix A for instructions on writing the output of `valgrind` to a log file instead of the terminal. You can install `valgrind` on your Pi (`sudo apt install valgrind`) to generate memory traces for testing.

Memory traces generated by `valgrind` have the following form:

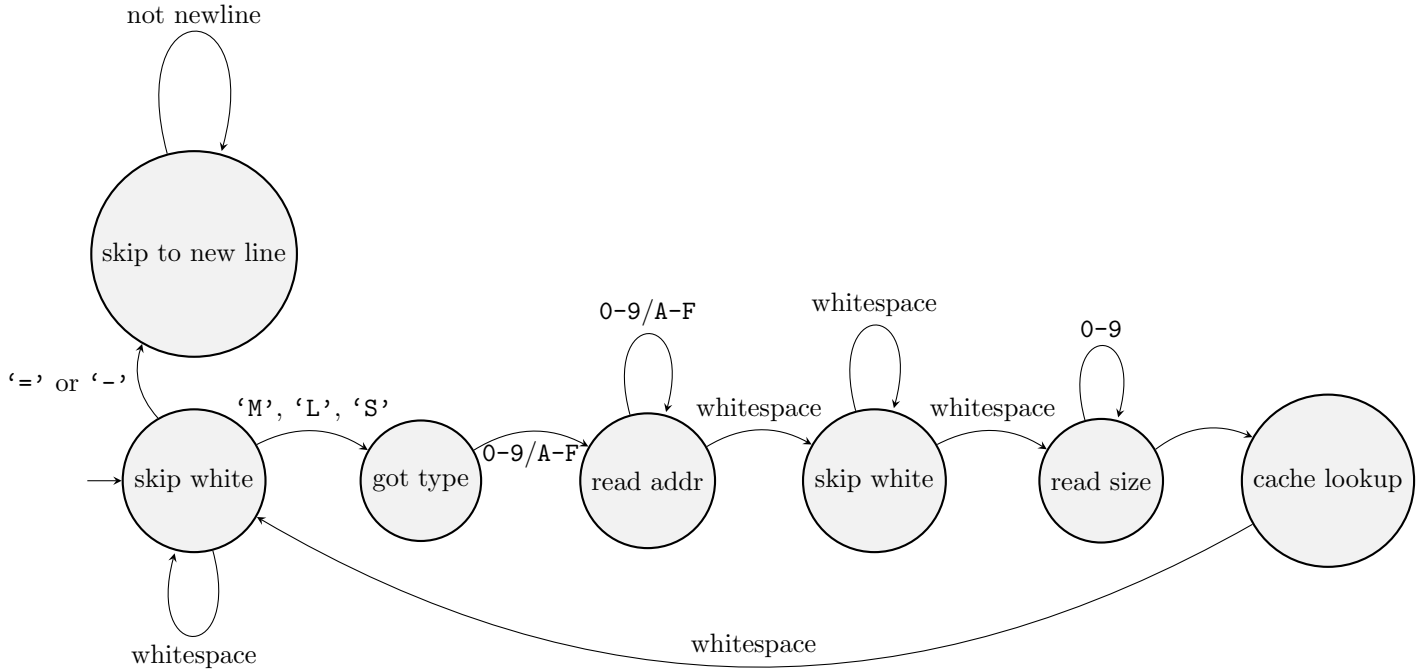
```
I 0400d7d4,8
M 0421c7f0,4
L 04f6b868,8
S 7ff0005c8,8
```

Access Type	Meaning
`I`	Instruction fetch
`M`	Data modify
`L`	Data load
`S`	Data store

Your cache simulator should ignore `I` accesses and only track access to the data cache (`L`/`S`/`M`). `valgrind` also generates some informational text at the beginning of the log, which can be treated as comments. Informational lines begin with `=` of `-` characters.

3.1 Parsing Memory Traces

`valgrind` memory traces are written in a formal grammar, which means that they can be processed by a finite state machine (FSM). An example FSM that you could use to parse memory transactions is drawn below.



4 Building the Simulator

In your simulator, you will need a C `struct` that simulates a cache line. Each cache line should have a valid and tag field. The full cache can be implemented as an array of cache line structs.

5 Homework Deliverables

1. Your project must include a working `Makefile`.
2. Your executable file must read a `valgrind` memory access log file and simulate the memory accesses on a cache. It should keep track of the total number of cache hits and misses.

3. After processing the log file, your program should print out the total number of cache hits and misses as well as the hit and miss rate as a percentage.

4. Your cache should have 128 lines and 8 bytes per line, the same as the example on the first page of this document.

```
Total Transactions: 2996974
Total Hits:         2011259 (67.109658%)
Total Misses:       985715 (32.890342%)
```

5. **Extra Credit:** make the cache geometry configurable at runtime. Use command line arguments to specify the number of lines and number of bytes per line in the cache. For example:

   `./cachesim -l 16 -b 128`

   Specifies 16 lines and 128 bytes per line. If you choose to implement the extra credit, you can take the cache geometry parameters from the commandl ine however you like—you don't have to use `-l XX` and `-b XX` to specify the cache size. Just make sure to tell us how to test it.

A Logging the Output of a Command to a File

You can redirect the output of a command to a file using the following syntax:

```
pi@raspberrypi ~ $ ls > out.txt
```

This takes the output of the `ls` command and redirects it to the file `out.txt`. Instead of printing the list of files in the current directory to the terminal, it will write the list to `out.txt`. You can also do this with more complex commands that take arguments on the command line. In the following example, I run `ls` with some options to print file modification times and ownership, logging the output to `out.txt`.

```
pi@raspberrypi ~ $ ls -lha > out.txt
```

A.1 `stdout` and `stderr`

If you try redirecting the output of `gcc` to a file, you will notice that it doesn't work very well:

```
pi@raspberrypi ~ $ gcc -v > vers.txt
Using built-in specs.
COLLECT_GCC=gcc
...
```

Why doesn't this cause the version information to be printed to `vers.txt`? The reason is that there are two different streams that you can use to print text to the terminal: `stdout` and `stderr`. `stdout` is the default pathway for printing text to the terminal. It is used by python's `print` and C's `printf`. When we redirect the output of a program to a file using the `>` operator, it is only `stdout` that gets redirected.

`stderr` is a different stream that some programs use to report error messages to the user. `gcc` uses `stderr` to print pretty much everything, including the version string. If you want to also print `stderr` to the same file as `stdout`, you have to tell the shell to do it:

```
pi@raspberrypi ~ $ gcc -v > vers.txt 2>&1
```

The last part of that command (`2>&1`) tells the shell to redirect the `stderr` to `stdout` so they both get printed to the same place. Then we redirect stdout to `vers.txt`, and nothing gets printed to the terminal.