4190.308 Computer Architecture, Fall 2015
Cache Lab: Understanding Cache Memories
Assigned: Thu, Nov 19, Due: Sun, Dec 13, 23:59
(hard deadline; no late submissions possible!)

1 Overview

This lab will help you to understand the operation of caches by implementing your own cache simulator.

2 Downloading the assignment

Download the handout file `cachelab.tgz` from eTL (`Computer Architecture` → `Projects` → `Cache Lab`) and save it in a directory of your choice. Next, extract the tarball by issuing the command

```
$ tar xvzf cachelab.tgz
```

This will create a directory called `cachelab` that contains a number of files. You will be modifying two files: `cache.c` and `cache.h`. The simulator main file that parses command line arguments and reads the memory trace is provided (and fully functional) in `cachesim.c`. To compile your simulator run

```
$ make clean
$ make
```

3 Writing a Cache Simulator

In this lab you will write a cache simulator in `cache.c/h` that takes a `valgrind` memory trace as input, simulates the hit/miss behavior of a cache memory on this trace, and outputs the total number of hits, misses, and evictions.

The input to the cache simulator are memory traces of real programs obtained by running `valgrind`. More about these traces and the trace generation can be found below.

The lab contains a reference implementation in `cachesim-ref`. The reference implementation supports six different cache line replacement strategies (round robin, random, LRU (least-recently-used), MRU (most-recently-used), LFU (least-frequently-used), and MFU (most-frequently-used)) and two write allocate strategies (write-allocate and no write-allocte). You don't have to implement all strategies (see below).

3.1 Usage

Running the reference implementation with `--help` produces the following help screen.

```
$ ./cachesim-ref -h
cachesim: a cache simulator.

Usage: $ cachesim <options>
where <options> is
  -h / --help                   Show this help screen.
  -v / --verbose                Be verbose while running.
  -c / --capacity <number>      Set the cache capacity.
  -b / --blocksize <number>     Set the block size.
  -w / --ways <number>          Set the number of ways.
  -r / --replacement <number>   Set the replacement strategy.
                                  0  round robin (default)
                                  1  random
                                  2  LRU (least-recently used)
                                  3  MRU (most-recently used)
                                  4  LFU (least-frequently used)
                                  5  MFU (most-frequently used)
  -W / --write <number>        Set the write strategy.
                                  0  write-allocate (default)
                                  1  no write-allocate
```

The capacity, blocksize, and the number of ways are mandatory parameters. The default replacement strategy (round robin) and write allocation policy (write-allocate) can be modified using the respective command line options.

The cache simulator reads its input from `stdin`. Use the pipe operator to cat a trace into the simulator as follows

```
$ cat traces/test.2.trace | ./cachesim-ref -c 256 -b 16 -w 1 -r 2
Cache configuration:
  capacity:           256
  blocksize:           16
  ways:                 1
  sets:                16
  tag shift:            8
  replacement:     LRU (least-recently used)
  on write miss:   write-allocate

Processing input...

Cache simulation statistics:
  accesses:              9
  hit:                   4
  miss:                  5
  evictions:             3
  miss ratio:     55.56%
```

The same example in verbose mode:

```
$ cat traces/test.2.trace | ./cachesim-ref -c 256 -b 16 -w 1 -r 2 -v
Cache configuration:
  capacity:           256
  blocksize:           16
  ways:                 1
  sets:                16
  tag shift:            8
  replacement:     LRU (least-recently used)
  on write miss:   write-allocate

Processing input...
R       10 [t:       0,s:  1]: miss alloc(free)
R       20 [t:       0,s:  2]: miss alloc(free)
W       20 [t:       0,s:  2]: hit
R       22 [t:       0,s:  2]: hit
W       18 [t:       0,s:  1]: hit
R      110 [t:       1,s:  1]: miss evict alloc(0)
R      210 [t:       2,s:  1]: miss evict alloc(0)
R       12 [t:       0,s:  1]: miss evict alloc(0)
W       12 [t:       0,s:  1]: hit


Cache simulation statistics:
  accesses:              9
  hit:                   4
  miss:                  5
  evictions:             3
  miss ratio:     55.56%
```

3.2 Implementing Your Cache Simulator

Your job for this lab is to implement a cache simulator that takes the same command line arguments and produces the identical output as the reference simulator.

You need to implement the following cache line replacement strategies:

- round-robin
- random (use the `rand()` function from `stdlib` to produce random numbers)
- LRU

The other strategies implemented by the reference simulator (MRU, LFU, MFU) are optional.

Your simulator should also support the following write allocation policies:

- write-allocate
- no write-allocate

You do not need to support verbose mode (`-v/--verbose` option), but we strongly encourage you to implement some form of feedback to debug your simulator.

The cache is implemented in `cache.c/h`. The data structures for the cache (`structs Cache`, `Set`, and `Line`) in `cache.h` have been prepared for you but you may need to add additional fields to manage the cache. The implementation of the functionality is in `cache.c`. You will find a number of functions with `TODO` markers followed by a short explanation that give you an idea what to do.

Programming Rules

- Your simulator must work correctly for arbitrary input parameters. Perform parameter validation and print an error if a parameter is invalid (for example, if the capacity is not a power of two). This also means that you will need to allocate storage for your simulator's data structures using the `malloc()` function. Type "man malloc" for information about this function.
- To receive credit for this lab, you must correctly update the variables `s_acces`, `s_hit`, `s_miss`, and `s_evict`. The main simulator file `cachesim.c` uses these variables to output the cache statistics.
- For this this lab, you should assume that memory accesses are aligned properly, such that a single memory access never crosses block boundaries. By making this assumption, you can ignore the request sizes in the `valgrind` traces.

3.3 Reference Trace Files

The `traces` subdirectory of this lab contains a collection of *reference trace files* that we will use to evaluate the correctness of your cache simulator. The trace files were generated by a Linux program called `valgrind` that records instruction and data read/write accesses of arbitrary programs.

The traces `test.?.trace` contain shorter traces that are useful when implementing and debugging your code. The four real-world traces `ls.1.trace`, `ls.2.trace`, `ls.3.trace.bz2`, and `cat.1.trace.bz2` are the result of running `ls` and `cat` on Linux machines. Two of the traces, `ls.3.trace` and `cat.1.trace` are so long that we provide them in compressed form.

Use the pipe operator to cat a trace into the simulator. For uncompressed traces, the command is (replace `test.4.trace` with any uncompressed trace file of your choice)

```
$ cat traces/test.4.trace | ./cachesim [cache options]
```

The compressed traces need to be decompressed before feeding them into the simulator. This can be acheived by running the decompressor `bunzip2` and piping the output directly into the simulator as follows

```
$ bunzip2 -c traces/cat.1.trace.bz2 | ./cachesim [cache options]
```

4 Evaluation

We will evaluate your simulator as follows

- code quality: 10 points
  You get full points if your code compiles without warnings, is indented correctly and reasonably commented.
- round-robin replacement policy: 20 points
- random replacement policy: 30 points
- LRU replacement policy: 30 points
- write-allocate/no write-allocate policyt: 10 points

We test your implementation with the trace files provided in `traces` and different cache parameters. Use the reference implementation to obtain the correct values.

We will give points for (partial) implementations that produce wrong results if you were on the right track, so do not remove code even if it may not work 100% correctly.

5 Handing in Your Work

Once you are ready to submit your work, enter your student ID and name in the file `STUDENT.ID`, then type

```
$ make submit
```

This will create a file `<studentid>.tgz` which you must send to the TA (`comparch@csap.snu.ac.kr`) by email.

Important: we must receive your submission by Sunday, December 13, 23:59, i.e., you cannot use any grace days for this lab. This is because we need to mark your labs so that we can compute your grades by Tuesday, December 15.