15-213 / 18-213 / 14-513 / 15-513 / 18-613, Spring 2021
Cache Lab: Understanding Cache Memories

Assigned: Thurs. March 11

Due: Tues. March 23, 11:59PM

Last hand-in: Fri. March 26, 11:59PM

Maximum grace days: 2

1 Introduction

This lab will help you understand the functioning of cache memories, and the impact that they can have on the performance of your C programs.

The lab consists of three parts. You will first write several traces to test the behavior of a cache simulator. Next, you will write a small C program (about 200-300 lines) that simulates the behavior of a hardware cache memory. Finally, you will optimize a matrix transpose function, with the goal of minimizing the number of cache misses.

This is an individual project. You should work on this lab on the Shark machines, which you can access via an SSH connection. You can find a list of all Shark machines at `https://www.cs.cmu.edu/~213/labmachines.html`.

1.1 Downloading the assignment

From this lab onwards, you will need to have a GitHub account to access lab materials. You can choose use an existing GitHub account with your personal email, or create a new GitHub account using your school email—either is fine. To create an account, visit `https://github.com/join`.

To get started, make sure you are signed into the GitHub account you want to use. Then click "Download handout" on Autolab, and follow the instructions. You will receive an email invitation to the repository within a few minutes. It will be located at:

`https://github.com/cmu15213s21/cachelab-s21-`*`yourgithubid`*

Once you accept the invitation, log onto a Shark machine, and change directories to a protected directory in AFS, such as `~/private/15213`. Then,"clone" (create a local copy of) your GitHub repository by entering the following command, substituting the URL of your specific repository:

```
$ git clone https://github.com/cmu15213s21/cachelab-s21-yourgithubid
```

This will create a directory containing the lab handout at `cachelab-s21-`*`yourgithubid`*.

1.2 Handout contents

Consult the file `README` for further descriptions of the handout files. For this lab, you will modify the following files:

```
traces/traces/tr1.trace
traces/traces/tr2.trace
traces/traces/tr3.trace
csim.c
trans.c
```

The `csim.c` and `trans.c` files will be compiled into C programs. You can do so by running `make`. Note that the file `csim.c` doesn't exist in your initial handout directory. You'll need to create it from scratch.

2 Traces: Writing Traces for a Cache Simulator (10 points)

2.1 Overview

A cache simulator is a program which simulates the behavior of a cache given a series of memory operations. The series of memory operations, loads and stores, is called a trace. In Part A of this lab, you will write your own cache simulator.

In this part of the lab, however, you will use a simulator provided by us to write traces that will produce specified cache behavior. That behavior could include, for example, a trace that generates a certain number of cache misses.

Your task is to write three traces based on the trace file format described in the next section. We have provided `example.trace`, which should give you an idea of how to begin.

Note: We have already created empty trace files for you. They can be found in the `traces/traces/` directory. The traces driver will read trace files from that location.

2.2 Trace File Format

The memory traces have the following form:

```
L 04f6b868,8
S 7ff0005c8,8
```

Each line denotes one memory access. The format of each line is: *Op Addr,Size*

- The *Op* field denotes the type of memory access: `L` represents a data load, and `S` a data store.
- The *Addr* field specifies the 64-bit hexadecimal memory address that will be accessed.
- The *Size* field specifies the number of bytes accessed.

For example, the first line above attempts to load 8 bytes starting from the address 0x04f6b868.

2.3 Specification

Your job for this part of the assignment is to write three traces, based on the format described above, which produce certain cache actions. This part should take no more than a few hours. For each of the following, you are to create a file with the specified file name in the traces directory.

- `tr1.trace`: The cache will be a direct-mapped cache with 8 sets and 16 byte blocks. Your trace should result in two hits and one eviction (any number of misses is okay). You are allowed a maximum of five operations.

- `tr2.trace`: The cache will be a 3-way set associative cache with two sets and 16 byte blocks. Your trace should have two hits and two misses (any number of evictions is okay). You are allowed a maximum of five operations.
- `tr3.trace`: The cache will be a 3-way set associative with four sets and 16 byte blocks. Your trace should result in exactly 5 hits, 4 misses, and one eviction. You are now allowed a maximum of ten operations.

The specifications are summarized in this table:

File Name	s	E	b	Requirements	Max Ops	Points
`tr1.trace`	3	1	4	2 hits, 1 eviction	5	3
`tr2.trace`	1	3	4	2 hits, 2 misses	5	3
`tr3.trace`	2	3	4	5 hits, 4 misses, 1 eviction	10	4

If a requirement is not specified, then that value can be anything (e.g. tr1.trace must have 2 hits and 1 eviction but any number of misses, dirty bytes, etc.).

2.4 Evaluation

We have provided several tools to help you test your traces.

- `traces-driver.py`: This is the same program that will run on autolab. To use this, run `./traces-driver.py`. This program will tell you the results of your traces and how many points you will receive. For more information, use the `-h` flag.
- `csim-ref`: This is the reference cache simulator that the driver will run. More information on this program is available later in this document.

To evaluate your code, we will run the reference simulator with the options specified in the table above, using the traces that you write. For each trace, you will get all of the specified points if your trace meets the requirements and no points otherwise.

3 Part A: Writing a Cache Simulator (60 points)

3.1 Overview

In Part A you will write a cache simulator that simulates the behavior of a cache, given a series of memory operations.

Your simulator should be able to simulate the behavior of a cache memory with arbitrary size and associativity. It should use the LRU (least-recently used) replacement policy when choosing which cache line to evict, and follow a write-back, write-allocate policy.

At the end of the simulation, it should output the total number of hits, misses, evictions, as well as the number of dirty bytes that have been evicted and the number of dirty bytes in the cache at the end of the simulation.

As a reminder, a dirty bit is a bit associated with each cache line. It is set whenever the payload of that block has been modified, but has not yet written back to main memory. A dirty byte is any payload byte whose corresponding cache block's dirty bit is set.

3.2 Reference simulator

We have provided you with the binary executable of a *reference cache simulator*, called `csim-ref`, that satisfies the specifications of the cache simulator assignment. The reference simulator takes the following command-line arguments:

```
Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

`-h`: Optional help flag that prints usage info

`-v`: Optional verbose flag that displays trace info

`-s <s>`: Number of set index bits ($S = 2^s$ is the number of sets)

`-E <E>`: Associativity (number of lines per set)

`-b <b>`: Number of block bits ($B = 2^b$ is the block size)

`-t <tracefile>`: Name of the memory trace to replay

The command-line arguments are based on the notation (s, E, and b) from page 617 of the CS:APP3e textbook. For example:

```
$ ./csim-ref -s 4 -E 1 -b 4 -t traces/csim/yi.trace
hits:4 misses:5 evictions:3 dirty_bytes_in_cache:32 dirty_bytes_evicted:16
```

The same example in verbose mode:

```
$ ./csim-ref -v -s 4 -E 1 -b 4 -t traces/csim/yi.trace
L 10,1 miss
L 20,1 miss
S 20,1 hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
L 12,1 miss eviction
S 12,1 hit
hits:4 misses:5 evictions:3 dirty_bytes_in_cache:32 dirty_bytes_evicted:16
```

3.3 Specification

In Part A you will write a cache simulator in `csim.c`. Notice that `csim.c` doesn't exist in your initial handout directory. You'll need to write it from scratch.

Your cache simulator should implement the `-s`, `-E`, `-b`, and `-t` flags in the same way as the reference simulator described above. The trace files will be in the same format as described in the first part of this assignment.

To receive credit, your code must call the `printSummary()` function, declared in `cachelab.h`, with the results of the simulation. This is the only result produced by your program that will be graded (so for example, your verbose output will not be graded).

```
void printSummary(const csim_stats_t *stats);
```

`printSummary` takes in a struct defined as follows, which contains the results of a cache simulation:

```
typedef struct {
  long hits;            /* number of hits */
  long misses;          /* number of misses */
  long evictions;       /* number of evictions */
  long dirty_bytes;     /* number of dirty bytes in cache at the end */
  long dirty_evictions; /* number of evictions of dirty lines */
} csim_stats_t;
```

Your code must obey the following programming rules:

- Include your name and Andrew ID in the header comment for `csim.c`.
- All of your code must be in `csim.c` and it must compile with the provided Makefile. Your `csim.c` file must compile without warnings in order to receive credit.
- For this lab, you can 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 memory traces.

- Your simulator must work correctly for caches of arbitrary size (i.e. arbitrary values of s, E, and b). Therefore, you may not allocate large data structures on the stack, since this will prevent large-sized caches from being created.

  Instead, for any data structures that could potentially be very large you will need to allocate storage on the *heap* using the `malloc` or `calloc` functions. Type "`man malloc`" for information about these functions.

  While your code will not be required to produce the correct output on unreasonably large values, your simulator should not crash in any case.

- The final simulator that you turn in should not print any extraneous or "debugging" output when verbose mode is not enabled. However, you are free to print whatever you like in verbose mode.

3.4 Evaluation

For Part A, we will run your cache simulator using different cache parameters and traces. There are ten test cases, each worth 5 points, except for the last case, which is worth 10 points:

```
$ ./csim -s 0 -E 1 -b 0 -t traces/csim/wide.trace
$ ./csim -s 2 -E 1 -b 2 -t traces/csim/wide.trace
$ ./csim -s 3 -E 2 -b 2 -t traces/csim/load.trace
$ ./csim -s 1 -E 1 -b 1 -t traces/csim/yi2.trace
$ ./csim -s 4 -E 2 -b 4 -t traces/csim/yi.trace
$ ./csim -s 2 -E 1 -b 4 -t traces/csim/dave.trace
$ ./csim -s 2 -E 1 -b 3 -t traces/csim/trans.trace
$ ./csim -s 2 -E 2 -b 3 -t traces/csim/trans.trace
$ ./csim -s 14 -E 1024 -b 3 -t traces/csim/trans.trace
$ ./csim -s 5 -E 1 -b 5 -t traces/csim/trans.trace
$ ./csim -s 5 -E 1 -b 5 -t traces/csim/long.trace
```

Note that the order of the `-s`, `-E`, `-b`, and `-t` parameters will vary during testing: your program must handle them when provided in any order.

You can use the reference simulator `csim-ref` to obtain the correct answer for each of these test cases. During debugging, use the `-v` option for a detailed record of each hit and miss.

We have provided you with an autograding program, called `test-csim`, which grades your program by comparing its output with that of the reference simulator. Be sure to compile your simulator before running the test:

```
$ make
$ ./test-csim
                                 Your simulator                        Reference simulator
Points ( s,    E,b)     Hits  Misses  Evicts D_Cache D_Evict     Hits  Misses  Evicts D_Cache D_Evict
     5 ( 0,    1,0)        1      18      17       1       6        1      18      17       1       6  traces/wide.trace
     5 ( 2,    1,2)        3      16      12       4      20        3      16      12       4      20  traces/wide.trace
     5 ( 3,    2,2)        6       3       0       0       0        6       3       0       0       0  traces/load.trace
     5 ( 1,    1,1)        9       8       6       4       8        9       8       6       4       8  traces/yi2.trace
     5 ( 4,    2,4)        4       5       2      32      16        4       5       2      32      16  traces/yi.trace
     5 ( 2,    1,4)        2       3       1      32      16        2       3       1      32      16  traces/dave.trace
```

```
 5 ( 2,   1,3)     167      71      67       8     264     167      71      67       8     264  traces/trans.trace
 5 ( 2,   2,3)     201      37      29      32     152     201      37      29      32     152  traces/trans.trace
 5 (14,1024,3)     215      23       0     120       0     215      23       0     120       0  traces/trans.trace
 5 ( 5,   1,5)     231       7       0     160       0     231       7       0     160       0  traces/trans.trace
10 ( 5,   1,5)  265189   21777   21745      96  556608  265189   21777   21745      96  556608  traces/long.trace
60
```

For each test, it shows the number of points you earned, the cache parameters, the input trace file, and a comparison of the results from your simulator and the reference simulator.

Each of the reported statistics is worth 1/5 of the points for each test case. For example, if a particular test case is worth 5 points, and your simulator outputs the correct number of hits and misses, but reports incorrect values for the other statistics, then you will earn 2 points.

3.5 Hints

Here are some hints and suggestions for working on Part A:

- To get started, you'll need to create a file called `csim.c` with an empty `main` routine. If you try running `make` before creating `csim.c`, it will fail with the message:

  ```
  make: *** No rule to make target ‘csim.c’, needed by ‘csim’.  Stop.
  ```

- The `printSummary` function is implemented in `cachelab.c`. In order to call it from `csim.c`, you will need to include the header file called `cachelab.h`:

  ```
  #include "cachelab.h"
  ```

- After calling `printSummary`, the `main` routine in `csim.c` should return a status of zero. If you return with a non-zero status value, the autograders will assume that there was an error.

- Do your initial debugging on the small traces, such as `traces/dave.trace`. You may also find the traces that you wrote earlier useful for testing basic functionality of your cache simulator — and you can even write new traces yourself.

- The reference simulator takes an optional `-v` argument that enables verbose output, displaying the hits, misses, and evictions that occur as a result of each memory access. Printing the same output as the reference simulator may help you debug by allowing you to directly compare the behavior of your simulator with the behavior of the reference simulator. However, you are not required to do so.

- We recommend that you use the `getopt` function to parse your command line arguments. You'll need the following header files:

  ```
  #include <stdlib.h>
  #include <unistd.h>
  #include <getopt.h>
  ```

See "`man 3 getopt`" for details.

- Each data load (`L`) or store (`S`) operation can cause at most one cache miss.
- Do not forget that the addresses in the trace are 64-bit **hexadecimal** memory addresses.

3.6 Memory errors

As part of style grading, we will check your cache simulator code for possible memory errors. Any such errors (besides resource cleanup) will result in a correctness deduction, which is counted separately from style. This check is NOT automatically performed by the provided autograder.

Valgrind is a tool that can detect various issues with the use of memory, such as:

1. Leaked memory (such as missing a call to `free`) — counted as resource cleanup
2. Out-of-bounds memory accesses (such as indexing past the end of an array)
3. Uninitialized memory usage (such as forgetting to initialize a local variable)
4. Incorrect calls to `malloc` or `free` (such as calling `free` twice)

You can run Valgrind on your `csim` program by prefixing an existing command with `valgrind -leak-check=full`, where we use the `-leak-check=full` flag to check for all possible memory leaks.

For example, a valid Valgrind command would be the following:

```
$ valgrind --leak-check=full ./csim -s 4 -E 2 -b 4 -t traces/csim/yi.trace
```

You should not run Valgrind on other programs, such as `test-trans` or `test-trans-simple`.

4 Part B: Optimizing Matrix Transpose (30 points)

4.1 Overview

In Part B you will write a transpose function in `trans.c` that uses as few clock cycles as possible, where the number of clock cycles is computed artificially using a cache simulator. The clock cycle computation captures the property that cache misses require significantly more clock cycles (100) than cache hits (4).

Let A denote a matrix, and $a_{i,j}$ denote the component at row i and column j. The *transpose* of A, denoted A^T, is a matrix such that $a^T_{i,j} = a_{j,i}$.

To help you get started, we have given you several example transpose functions in `trans.c` that compute the transpose of $N \times M$ matrix A and store the results in $M \times N$ matrix B. An example of one such function is:

```
void trans(size_t M, size_t N, const double A[N][M], double B[M][N], double *tmp);
```

Argument `tmp` is a pointer to an array of 256 elements that can be used to hold data as an intermediate step between reading from `A` and writing to `B`.

The example transpose functions are correct, but they have poor performance, because the access patterns result in many cache misses, resulting in a high number of clock cycles.

4.2 Specification

Your job in Part B is to write a similar function, called `transpose_submit`, that minimizes the number of clock cycles across different sized matrices:

```
void transpose_submit(size_t M, size_t N, const double A[N][M], double B[M][N],
                      double *tmp);
```

Do *not* change the description string ("`Transpose submission`") for your `transpose_submit` function. The autograder uses this string to determine which transpose function to evaluate for credit.

Your code must obey the following programming rules:

- Include your name and Andrew ID in the header comment for `trans.c`.
- All of your code must be in `trans.c`, and it must compile with the provided Makefile. Your code in `trans.c` must compile without warnings to receive credit.
- You may use helper functions. Indeed, you will find this a useful way to structure your code. You may also use recursion if you find it to be useful.
- Your transpose function may not modify array `A`. You may, however, read and/or write the contents of `B` and `tmp` as many times as you like.

- You may not store any array data (i.e. floating-point data) outside of `A`, `B`, and `tmp`. This includes any other local variables, structs, or arrays in your code.
- You may not make out-of-bounds references to any array.
- You are NOT allowed to use any variant of `malloc`.
- Since our style guidelines prohibit the use of "magic numbers," you should refer to the maximum number of elements in `tmp` with the compile-time constant `TMPCOUNT`.
- These restrictions apply to *all* functions in your `trans.c` file, not just those that are called as part of the official submission.
- You may customize your functions to use different approaches depending on the values of M and N. Indeed, you will find this necessary to achieve the required performance objectives.

4.3 Registering transpose functions

You can register up to 100 versions of the transpose function in your `trans.c` file. Register a particular transpose function with the autograder by making a call of the form:

```
registerTransFunction(trans_simple, "A simple transpose");
```

in the `registerFunctions` routine in `trans.c`. At runtime, the autograder will evaluate each registered transpose function and print the results.

One of the registered functions must be the `transpose_submit` function, which is the one that will be submitted for credit, as mentioned above:

```
registerTransFunction(transpose_submit, SUBMIT_DESCRIPTION);
```

See the default `trans.c` function for an example of how this works.

4.4 Evaluation

4.4.1 Correctness

If any of the programming rules are violated, you will receive **no credit** for Part B. Although the compiler is configured to detect some violations of these guidelines automatically, your code may also be manually checked when style points are assigned.

Additionally, for Part B, we will evaluate the correctness of your `transpose_submit` function on ten different matrix sizes. You will receive **no credit** for part B if your code gives incorrect results for any of these.

To evaluate the correctness of your code, we have provided you with a program, `test-trans-simple.c`, that tests the correctness of each of the transpose functions that you have registered for a specific matrix size. You can run it as follows:

```
$ make
$ ./test-trans-simple -M 32 -N 32
Function 0 (Transpose submission): Correct
Function 1 (Simple row-wise scan transpose): Correct

Summary for official submission (func 0): correctness=1
```

In this example, we have registered two different transpose functions in `trans.c`. The `test-trans-simple` program tests each of the registered functions, displays the results for each, and extracts the results for the official submission.

The `test-trans-simple` program is for your use only, to assist in debugging. It will not be used by the autograder: instead, the `test-trans` program (described in the next section) will be used to check performance and correctness simultaneously.

However, the `test-trans-simple` program is compiled with AddressSanitizer enabled, to assist you in detecting out-of-bounds array accesses. If you have questions about any error messages you receive from it, don't hesitate to ask us for help.

4.4.2 Performance

We will evaluate the performance of your transpose function on two different-sized matrices:

- 32×32 ($M = 32$, $N = 32$)
- 1024×1024 ($M = 1024$, $N = 1024$)

We have provided you with an autograding program, called `test-trans.c`, that tests the correctness and performance of each transpose function you have registered with the autograder.

For each of the matrix sizes above, the performance of your `transpose_submit` function is evaluated by using LLVM-based instrumentation to extract the address trace for your function, and then using the reference simulator to replay this trace on a cache with parameters $s = 5$, $E = 1$, $b = 6$.

However, for the 1024x1024 matrix, we will be evaluating the performance of your transpose function on the Haswell L1 cache with cache parameters $s = 6$, $E = 8$, $b = 6$. You can test with these parameters by passing the `-l` flag to `test-trans`.

For example, to test your registered transpose functions on a 32×32 matrix, rebuild `test-trans`, and then run it with the appropriate values for M and N:

```
$ make
$ ./test-trans -M 32 -N 32
Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=6)
func 0 (Transpose submission): hits:868, misses:1180, evictions:1148,
clock_cycles:121472
```

```
Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=6)
func 1 (Row-wise scan transpose): hits:868, misses:1180, evictions:1148,
clock_cycles:121472

Summary for official submission (func 0): correctness=1 cycles=121472

$ ./test-trans -M 1024 -N 1024 -l
...
```

Using the reference cache simulator, each transpose function will be assigned some number of clock cycles m. A cache miss is worth 100 clock cycles, while a cache hit is worth 4. Your performance score for each matrix size will scale linearly with m, up to some threshold. The scores are computed as:

- 32×32: 20 points if $m <$ 36,000, 0 points if $m >$ 45,000
- 1024×1024: 10 points if $m <$ 35,100,000, 0 points if $m >$ 45,000,000

For example, a solution for the 32×32 matrix with 1764 hits and 284 misses ($m = 1764 \cdot 4 + 284 \cdot 100 = 35456$) would receive 20 of the possible 20 points.

You can optimize your code specifically for the two cases in the performance evaluation. In particular, it is perfectly OK for your function to explicitly check for the matrix sizes and implement separate code optimized for each case.

4.5 Hints

Here are some hints and suggestions for working on Part B.

- Since your transpose function is being evaluated on a direct-mapped cache, conflict misses are a potential problem, both within the individual matrices, between them, and between the matrices and the temporary data. Think about the potential for conflict misses in your code, especially along the diagonal. Try to think of access patterns that will decrease the number of these conflict misses. This will in turn lower the number of clock cycles required.
- You are guaranteed that matrices `A` and `B`, and temporary storage `tmp` all align to the same positions in the cache. That is, if a_A, a_B, and a_t are the starting addresses of `A`, `B`, and `tmp`, respectively, then $a_A \bmod C = a_B \bmod C = a_t \bmod C$, where C is the cache size (in bytes). Also, these all begin on a cache-block boundary. That is, $a_A \bmod B = a_B \bmod B = a_t \bmod B = 0$, where $B = 2^b$ is the block size.
- It is not likely that you will want to use 256 temporaries in any of your transpose routines. However, having this many allows you to strategically choose which ones to use in order to avoid conflicts with the elements of `A` and `B` you are reading and writing.

- Blocking is a useful technique for reducing cache misses. See `http://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf` for more information. You will need to experiment with a number of different blocking strategies.

5 Style grading (4 points)

From this lab onwards, we will begin to assign a style grade for all code you submit in this course. Information about the style guidelines can be found on the course website at `https://www.cs.cmu.edu/~213/codeStyle.html`.

There are 4 points for coding style in this lab. These will be assigned by the course staff after the submission deadline, based on the code in your Autolab submission.

5.1 Code formatting

Your code for all labs must be formatted correctly to receive points on Autolab. For formatting your code, we require that you use the `clang-format` tool. To invoke it, run `make format`. You can modify the `.clang-format` file to reflect your preferred code style.

For cachelab, the formatting requirement applies to all C files that you submit, i.e. both `csim.c` and `trans.c`.

5.2 Version control

Starting from this lab, you must commit your code regularly using Git. This allows you to keep track of your changes, revert to older versions of your code, and regularly remind yourself of what you changed and why you made those changes. For specific guidelines on Git usage, see the style guideline.

Remember that you must always **push your commits to GitHub** for them to be counted, since that is where we will obtain your commit history from.

5.3 Code style

Out of your style grade for this lab, **4 manually-graded points** will be awarded based on the `csim.c` file *only*. Make sure to read the entire style guideine. However, here are some points that you should keep in mind for cachelab in particular:

- **File comment.** Your `csim.c` file **must** begin with a file comment that gives an overview of what the file does, and of relevant design decisions. For example, you might want to describe the data structures used to implement your cache simulator.
- **Other comments.** In addition to the file comment, use comments to describe what each function does. Additionally, use inline comments to explain any particularly tricky code. Finally, describe any structs, enums, or global variables that you declare. Make sure to review the course style guide for more details.
- **Modularity.** Your code should be decomposed into functions in a way that makes the code easier to read. Aim for each function to have a single, well-defined purpose that can be described in a sentence (which should be in the function comment!).

In particular, avoid writing extremely long functions and duplicating large amounts of code. Instead, think about how you can write helper functions to make your code more modular.

- **Magic numbers.** Avoid sprinkling your code with numeric constants. Instead, declare such constants at the top of the file by using `#define` or by using `const` variables. This helps to make your code more extensible in the future.

- **Readability.** For instance, choose appropriate variable and function names, and avoid including commented-out code in your final submission. Paying attention to these things can go a long way towards making your code more readable.

- **Error checking.** You must consider the possibility that a library function that you call will fail. Whatever the case, your program should make sure that it never crashes: all errors should be detected and handled in an appropriate manner — see the style guidelines for more.

  You are welcome to use various techniques to handle errors, such as the `xmalloc` function from 15-122, or the `Malloc` function from the textbook. However, you must implement those functions yourself, and you yourself are ultimately responsible for ensuring that the behavior of your program is correct.

- **Resource cleanup.** Your code must release **all** allocated resources before it exits. In particular, this includes any allocated memory or opened files. This can be automatically checked by Valgrind (see §3.6).

  It is acceptable to avoid cleaning up resources if the program needs to terminate abnormally, if you decide it would be otherwise too difficult to do so. However, you should document and briefly justify this decision in your code.

Finally, don't forget that your cache simulator will also be tested for memory errors with Valgrind, as described in §3.6. Any such errors (excluding resource cleanup) will count as correctness deductions, which is counted separately from style.

6 Putting it all Together

6.1 Scoring

Cachelab is worth 5% of your final grade in this course. The maximum score for this lab is 104 points, which will be assigned as follows:

Traces assignment (§2)	10 points
Part A: Cache Simulator (§3)	60 points
Part B: Matrix Transpose (§4)	30 points
Style grading (§5)	4 points
Total	**104 points**

6.2 Driver program

We have provided you with a *driver program*, called `./driver.py`, that performs a complete evaluation of your traces, simulator and transpose code. To run the driver, type:

```
$ ./driver.py
```

The driver uses `traces-driver.py` to evaluate your traces, `test-csim` to evaluate your simulator, and uses `test-trans` to evaluate your submitted transpose function for correctness (ten matrix sizes) and performance (two matrix sizes). Then it prints a summary of your results and the points you have earned. This is the same program that Autolab uses when it autogrades your handins.

6.3 Handin

To receive a score, you should upload your submission to Autolab. The Autolab servers will run the same driver program that is provided to you, and record the score that you receive. You may handin as often as you like until the due date.

There are two ways you can submit your code to Autolab.

1. Running the `make` command will generate a tar file, `cachelab-handin.tar`. You can upload this file to the Autolab website.
2. If you are running on the Andrew Unix or Shark machines, you can submit the tar file directly from the command line as follows:

   ```
   $ make submit
   ```

IMPORTANT: Do not assume your submission will succeed! You should ALWAYS check that you received the expected score on Autolab. You can also check if there were any problems in the autograder output, which you can see by clicking on your autograded score in blue.