18-213 / 18-613, Summer 2025
Cache Lab: Understanding Cache Memories
Assigned: Thursday, Jun 5
Due: Thursday, Jun 12, 11:59PM
Last hand-in: Thursday, Jun 19, 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. You will also learn how to write C programs that can be run from the Unix command line.

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/~18213/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 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/18-x13/cachelab4-`*`yourgithubid`*

Once you accept the invitation, log onto a Shark machine, and change directories to a protected directory in AFS, such as `~/private/18213`. 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/18-x13/cachelab4-yourgithubid
```

This will create a directory containing the lab handout at `cachelab4-`*`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 and check it in (`git add csim.c`).

2 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 (loads and stores). The series of memory operations is called a trace.

In this part of the lab, you will write three traces that should produce specific types of cache behavior. We have provided you with a simulator (`csim-ref`) to test the traces with, but you must write them in the format it expects.

Note: We have already created empty trace files for you. They can be found in the `traces/traces/` subdirectory of your lab handout. The test driver will read trace files from that location. This directory also contains an example trace file, `example.trace`, which should give you an idea of how to begin.

2.2 Trace File Format

`csim-ref` expects traces to be text files with one memory operation per line. Each line must be in the format: *Op Addr,Size*.

Op denotes the type of memory access. It can be either `L` for a load, or `S` for a store.

Addr gives the memory address to be accessed. It should be a 64-bit hexadecimal number, *without* a leading `0x`.

Size gives the number of bytes to be accessed at *Addr*. It should be a small, positive decimal number.

Here is a short example:

```
L 4f6b868,4
S 7ff005c8,8
```

This trace instructs the simulator to simulate a load of four bytes from address `0000 0000 04f6 b868` and then a store of eight bytes to address `0000 0000 7ff0 05c8`.

Note: Trace files do not specify the actual *values* being loaded or stored, only the number of bytes. This is because the simulator does not need to know the values. (Why?)

Note: The reference simulator only understands trace files in the format shown above. There is no way to write comments in trace files. The only valid *Op* codes are uppercase `L` and uppercase `S`. The comma between *Addr* and *Size* is required, as is the space between *Op* and *Addr*. There should be *no* spaces on either side of the comma.

2.3 Traces To Be Written

Your job for this part of the assignment is to write three traces, in 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

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.

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

`traces-driver.py`: This is the program Autolab will use to evaluate this part of the assignment. If you run it yourself, it will tell you the results of your traces and how many points you will receive. To run it use the command `./traces-driver.py`. 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.

3 Writing a Cache Simulator (60 points)

3.1 Overview

In the second part of this lab, you will write a cache simulator that simulates the behavior of a cache, given a trace in the same format you used above.

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, and 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 *byte* is any byte in a cache block that has been modified but not yet written back to main memory. All of the bytes in a modified cache block are considered to be dirty even if that specific byte was not modified. The number of dirty bytes in the cache and the number of dirty bytes that have been evicted are therefore always a multiple of the cache block size. A dirty *bit*, on the other hand, is a bit associated with each cache line that tracks whether the block held by that cache line has been modified but not yet written back to main memory.

3.2 Writing a Command-Line Tool from Scratch

You may or may not have already learned how to write *command-line tools*: programs designed to be invoked from the Unix shell and vary their behavior based on *arguments* they receive on startup. The cache simulator you will write in this part of the lab is expected to be just such a tool. Because you might never have needed to do this before, in this lab we ask you to write the cache simulator from scratch. The file `csim.c` does not exist in the starting repository; you must create it and write everything that goes in it.

The standard C runtime library contains many functions that are helpful when writing command-line tools. Functions that are useful for this lab, that you may not have needed before, include `getopt`, `strtoul`, `strtok`, `fopen`, `fclose`, and `fgets`. You can read more about these functions in their *man pages*: type "`man 3 function-name`" at a shell prompt. The 3 means "show me the documentation for a C function, not a shell command." If you get an error saying "No manual entry for *function-name* in section 3", or if the documentation that comes up doesn't seem to be about a C function, try changing the 3 to a 2. (For historical reasons, the C library's documentation is split across two "sections.")

Man pages can also be read on the Web in several places: we recommend looking first at https://www.man7.org/linux/man-pages/index.html which has (a newer edition of) the same man pages found on the sharks. Another well-written collection of man pages can be found at https://man.openbsd.org/. These were written for OpenBSD, not Linux, but the C library works mostly the same.

If you would prefer to read a manual organized by topic, rather than with a page for each individual function, we recommend the GNU C Library Reference Manual, which is also on the Web: the table of contents is at https://www.gnu.org/software/libc/manual/html_node/index.html#SEC_Contents and an index of individual functions is at https://www.gnu.org/software/libc/manual/html_node/Function-Index.html.

The cache simulator you used in the first part of the lab (`csim-ref`) can be used as a reference for correct command-line behavior. If you run it with no arguments, it prints the following message:

```
Mandatory arguments missing or zero.
Usage: ./csim-ref [-v] -s <s> -E <E> -b <b> -t <trace>
       ./csim-ref -h

    -h          Print this help message and exit
    -v          Verbose mode: report effects of each memory operation
    -s <s>      Number of set index bits (there are 2**s sets)
    -b <b>      Number of block bits (there are 2**b blocks)
    -E <E>      Number of lines per set (associativity)
    -t <trace>  File name of the memory trace to process

The -s, -b, -E, and -t options must be supplied for all simulations.
```

This message tells you that `csim-ref` requires arguments to do anything useful, and then tersely summarizes these arguments. The second and third lines of the message show two different *valid* command line invocations

of `csim-ref`. You can supply all four of the `-s`, `-b`, `-E`, and `-t` options, each of which takes a value, and optionally also the `-v` option, which doesn't take a value. (The options are listed in one particular order, but `csim-ref` will accept them in any order, as long as each option is immediately followed by its value, if any.) Or you can supply the `-h` option, which doesn't take a value, and nothing else.

The message goes on to tell you what each of the options mean. `-h` makes `csim-ref` print the same message that it prints if you don't give it any arguments (but without the initial "Mandatory arguments missing or zero" error). `-v` makes `csim-ref` be *verbose* and tell you details of what it is doing as it does it. These are both *common* options: many Unix command line tools interpret `-h` and `-v` the same way `csim-ref` does.

`-s`, `-b`, and `-E` specify the size parameters of the cache to be simulated. (The letters were chosen to match the *s*, *b*, and *E* notation used on page 617 of the CS:APP3e textbook.) The value for each is described as a "number." `-t` specifies the file name of the memory trace to process. These options are specific to `csim-ref`; the same letter codes might mean something completely different to a different command line program.

Finally, `csim-ref` reminds you that you must supply all four of the `-s`, `-b`, `-E`, and `-t` options "for all simulations"—that is, `csim-ref` will only run a simulation if all four of these options are given. This is abnormal for Unix command line tools: "options" are supposed to be *optional*. But `csim-ref` really does need to know all four of these options' values to run a simulation.

Incidentally, '`2**n`' is a conventional way of writing '2^n' when you can't use superscripts. Sometimes people use '`2^n`' for this, but in C that means XOR, so the authors of `csim-ref` didn't want to use it here.

3.2.1 Processing Command Line Arguments

Your first task in this part of the lab is to write a program that accepts the same set of command line arguments that `csim-ref` does. Command line arguments are supplied to C programs as the `argv` array argument to `main`, which has `argc` elements.

```
#include <stdio.h>
int main(int argc, char **argv)
{
    for (int i = 0; i < argc; i++) {
        printf("Command line argument %d is %s\n", i, argv[i]);
    }
    return 0;
}
```

(Element 0 of `argv` is special. What does it contain?)

You need to find the various options, and their values, in the array, turn the numeric values from strings into machine numbers, and put all the values into variables. Don't worry about *doing* anything with these variables yet, except maybe print them back out to make sure you got it right. You don't have to duplicate `csim-ref`'s help message exactly, but you should print *something* useful when `-h` is the only argument.

You also need to detect *incorrect* invocations, such as:

- Not all of `-s`, `-b`, `-E`, and `-t` were supplied.
- The value for `-s`, `-b`, `-E`, or `-t` is missing.
- The value for `-s`, `-b`, or `-E` is not a positive integer, or is too large to make sense. (How many bits are there in an address? What does that tell you about the sum $s + b$?)
- An option (argument beginning with '`-`') was given that isn't in the list of recognized options.
- Arguments were given that aren't options or values for options.

Play around with incorrect invocations of `csim-ref`; you will find that it gives errors for all of the above cases and perhaps others.

As a reminder, the C library functions `getopt` and `strtoul` may be helpful for this first task.

3.3 Parsing trace files

Your second task is to write a *parser* for memory access trace files like the ones you wrote in the first part of the lab, and arrange to run this parser on the trace file specified on the command line. As with the command line arguments, don't worry about *doing* anything with the data you parse yet, except maybe print it back out to make sure you got it right.

Memory access traces are an example of a *line-oriented* file format. Refer to Section 2.2 for a detailed description of trace file syntax. You do not have to be flexible when parsing—for instance, it's fine to insist that there be exactly one space after the *Op* code. (However, it might be nice to be a *little* flexible. What does `csim-ref` do?)

You should detect and reject input that is completely incorrect, such as:

- Lines that don't have all three of *Op Addr*,*Size*.
- Lines with trailing junk after *Size*.
- Lines where *Op* is neither '`L`' nor '`S`'.
- Lines where *Addr* or *Size* is out of range or not a number at all.

Line-oriented files are relatively easy to parse using standard C library functions. You can use `fopen` and `fclose` to access the file, `fgets` to read each line from the file, and, again, `strtoul` to turn strings into machine numbers. The `strtok` function may be useful in splitting up the line into fields, but it's not strictly necessary. Remember that a string in C is just an array of characters: you can write things like '`s[0] == 'L'`' to find out if the first character in a string is '`L`'.

Resist the temptation to use `fscanf` or `sscanf`. These functions may, on first glance, appear to save you a lot of trouble, but it is *more* difficult to detect incorrect input with them than with `strtok` and `strtoul`.

A skeleton of your parser function might look something like this:

```
/** Process a memory-access trace file.
 *
 *  @param trace   Name of the trace file to process.
 *  @return        0 if successful, 1 if there were errors.
 */
int process_trace_file(const char *trace)
{
    FILE *tfp = fopen(trace, "rt");
    if (!tfp) {
        fprintf(stderr, "Error opening '%s': %s\n",
                trace, strerror(errno));
        return 1;
    }

    char linebuf[LINELEN];  // How big should LINELEN be?
    int parse_error = 0;
    while (fgets(linebuf, LINELEN, tfp)) {
        // Parse the line of text in 'linebuf'.
        // What do you do if the line is incorrect?
        // What do you do if the line is longer than
        //   LINELEN-1 chars?
    }
    fclose(tfp);
    // Why do I return parse_error here and not 0?
    return parse_error;
}
```

3.4 Simulating a cache

Your third and final task in this part of the lab is to take the parameters you receive on the command line and the data you parse from trace files and use them to simulate accesses to memory via a cache. You can assume that `csim-ref` does this simulation correctly. For example, if you run `csim-ref` on `yi.trace` with $s = 4$, $E = 1$, and $b = 4$, you get a report saying that there were 4 cache hits, 5 misses, and 3 evictions, and, at the end of the simulation, there were 32 dirty bytes still in the cache and a total of 16 dirty bytes had been evicted back to RAM.

```
$ ./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
```

Your simulator must produce the same report, in the same format. In the normal mode (without `-v`), it should not produce any other output. We have provided you a helper function that prints the report in the right format. It is declared in `cachelab.h`, along with the `struct` it takes as an argument.

```
typedef struct {
  unsigned long hits;            /* number of hits */
  unsigned long misses;          /* number of misses */
  unsigned long evictions;       /* number of evictions */
  unsigned long dirty_bytes;     /* number of dirty bytes in cache
                                    at end of simulation */
  unsigned long dirty_evictions; /* number of bytes evicted
                                    from dirty lines */
} csim_stats_t;
void printSummary(const csim_stats_t *stats);
```

You can create one of these structures, update it as you go through the simulation, and pass it to `printSummary` at the end of execution.

If you run the same `csim-ref` command, but add `-v` to enable verbose mode, it will print the same report at the end, but it will also print a line for each operation in the trace, describing its effects:

```
$ ./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
```

You do not have to implement verbose mode yourself, but it can be very helpful for debugging. If you do implement it, the output does not have to match what `csim-ref` prints, but it must still end with the same report that would have been printed without `-v`.

You may assume that all of the memory accesses in the traces are properly aligned and therefore each memory access only affects one cache block. (Fun fact: this means you can ignore the request sizes in the memory traces. Do you see why?)

3.5 Evaluation

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
```

Each of the reported statistics (hits, misses, evictions, dirty bytes, evicted dirty bytes) is worth ⅕ of the points for the 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 for that test.

You can use `csim-ref` to obtain the correct statistics for each test case. During debugging, use the `-v` option for a detailed record of each hit and miss.

We have provided you with a program called `test-csim` that compares your simulator's output with `csim-ref`'s output. 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  wide.trace
     5 ( 2,   1,2)       3      16      12       4      20       3      16      12       4      20  wide.trace
     5 ( 3,   2,2)       6       3       0       0       0       6       3       0       0       0  load.trace
     5 ( 1,   1,1)       9       8       6       4       8       9       8       6       4       8  yi2.trace
     5 ( 4,   2,4)       4       5       2      32      16       4       5       2      32      16  yi.trace
     5 ( 2,   1,4)       2       3       1      32      16       2       3       1      32      16  dave.trace
     5 ( 2,   1,3)     167      71      67       8     264     167      71      67       8     264  trans.trace
     5 ( 2,   2,3)     201      37      29      32     152     201      37      29      32     152  trans.trace
     5 (14,1024,3)     215      23       0     120       0     215      23       0     120       0  trans.trace
     5 ( 5,   1,5)     231       7       0     160       0     231       7       0     160       0  trans.trace
    10 ( 5,   1,5)  265189   21777   21745      96  556608  265189   21777   21745      96  556608  long.trace
    60
```

For each test, `csim-ref` 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.

Your code must obey the following programming rules. Some of these will be enforced automatically and others will be checked by your TA when they review your code.

- All of your code must be in `csim.c` and it must compile with the provided Makefile. For this lab, you may *not* create any header files yourself.
- You will be graded on the code you have checked in and pushed to GitHub, not the code in your home directory on the sharks. Do not forget to `git add csim.c`, `git commit`, and `git push` regularly!
- Your code will be manually reviewed for good style and coding practices; see Section 5 for details.
- All of your code must compile without warnings.
- Your simulator must accept the `-s`, `-E`, `-b`, and `-t` options in *any order*. (This will happen naturally if you use `getopt` to process command line arguments.)
- Your simulator must 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.
- After it is finished, your simulator must exit with a meaningful *exit code*. This means returning `0` from `main` (or, equivalently, calling `exit(0)`) if it has successfully simulated a trace, and returning `1` from

`main` (or, equivalently, calling `exit(1)`) if it encountered any sort of error (incorrect command line arguments, syntax errors in the traces, etc.) If the autograder deducts points for no visible reason, the problem is likely to be that you got this wrong.

- Your simulator must not crash, even if given an invalid trace file or nonsensical command line arguments.
- Your simulator must handle arbitrarily large caches (i.e. arbitrarily large values of s, e, and b, as long as $s + b$ is still in bounds). Remember that local variables are allocated from the stack, and the stack is limited in size. Data structures that can be very large should be allocated using `malloc` or `calloc`.

3.6 Hints

Here are some hints and suggestions for working on the simulator:

- 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.
- In verbose mode, 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 this.
- To use any C library function, you must include its header file. Some library functions need you to include more than one header file. The set of required header files for each function is listed at the very top of its man page (the `SYNOPSIS` section).
- `printSummary` is a function we provided you, not a C library function; it does not have a manpage. It is documented in this writeup. To use it you must include `cachelab.h`.
- Each data load (`L`) or store (`S`) operation can cause at most one cache miss. (Why?)
- Do not forget that the addresses in the trace are 64-bit **hexadecimal** memory addresses.

3.7 Memory errors

As part of code review, we will check your cache simulator code for possible memory errors. Any such errors (besides resource cleanup) will result in a correctness deduction. 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 by prefixing any `csim` command with `valgrind -leak-check=full`. The `-leak-check=full` flag means 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 Optimizing Matrix Transpose (30 points)

4.1 Overview

In the third part of this lab, 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, 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 this part of the lab 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, 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 your implementation. Although the compiler is configured to detect some violations of these guidelines automatically, your code will also be manually checked during code review.

Additionally, we will evaluate the correctness of your `transpose_submit` function on ten different matrix sizes. You will receive **no credit** for `transpose_submit` 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, two transpose functions are registered by `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 matrix transposition.

- 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 Code review (4 points)

From this lab onwards, your TA will manually review your code for good style and programming practices. Information about the style guidelines can be found on the course website at https://www.cs.cmu.edu/~18213-s24/codeStyle.html.

Code review is worth 4 points in this lab. These points 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 Good style and coding practice

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 comment that gives an overview of what the file does, and of relevant design decisions. When writing this comment, imagine that you are introducing your program to someone who has never used it before. What would they want to know about it? Some suggestions:

- How do you *use* this program? What does it do? What input does it need? What command line arguments does it expect? (Think about what we have told you about `csim-ref` in this writeup.)
- What would you need to know in order to modify this program? How do its data structures work? Is there code with subtle or tricky requirements, that would be easy to break by accident, perhaps by modifying something else?
- Why is the program designed the way it is, and not some other way?

Other comments. Each function, each `struct` declaration, and each global variable should also have a comment immediately above it that describes what it is for and when it should be used. Sometimes this can be very short:

```
/** True if the simulator should report the effect of each
    memory operation on the cache, as it processes the trace.  */
static bool verbose_mode = false;
```

Other times (especially for important functions and structs) you might need to write several paragraphs. When writing these comments, imagine yourself describing the code to someone who wants to use it but does not know *when* it would be appropriate.

This is an *unhelpful* function comment:

```
/** Enable verbose mode. */
static void enable_verbose_mode(void)
{ verbose_mode = true; }
```

It communicates nothing that the reader did not already know from looking at the name and the body of the function. A more helpful comment for a tiny function like this, might be something like:

```
/** @brief Enable verbose mode.
 *
 * See the declaration of 'verbose_mode' for what output
 * is produced in this mode.  Related knobs are 'debug_mode'
 * and 'statistics_mode'.
 */
static void enable_verbose_mode(void)
{ verbose_mode = true; }
```

Comments *inside* functions should be reserved for explaining the *reasons* for particularly tricky bits of code. Resist the temptation to translate each line of code into English; this can be helpful when you are *writing* the code to keep it straight in your own head, but it is worse than useless to future readers.

Good example:

```
int crypt (const char *phrase, const char *setting)
{
    /* Do these strlen() calls before reading prefixes of
       'phrase' or 'setting', so we get a predictable crash
       if they are not valid strings.  */
    size_t phr_size = strlen (phrase);
    size_t set_size = strlen (setting);
```

Bad example:

```
/* Return immediately if we have nothing to do. */
if (len == 0)
    return;
```

Consult 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 §3.7).

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 §3.7. 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:

Writing traces (§2)	10 points
Cache simulator (§3)	60 points
Matrix transpose (§4)	30 points
Code review (§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 and have already authorized the CLI, 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.