Project description (CS-453)

Sébastien Rouault

November 5, 2020

Project forum: `https://moodle.epfl.ch/mod/forum/view.php?id=1085060`

Project contact: `sebastien.rouault@epfl.ch`

Contents

1 Software transactional memory

Your goal is to implement a *software transactional memory* library.

The project repository (see Section 2.1) contains a very simple reference implementation of such a library, along with the skeleton for another implementation: the one you'll have to (design and) code.

The project repository includes the source code for, and a *Makefile* to run the program that will evaluate your implementation on the evaluation server (provided by the *Distributed Computing Laboratory*).

But first, let's informally describe what a (software) transactional memory is, and why it can be useful.

1.1 Gentle introduction

Let me introduce you to Alice. Alice runs a bank. As for any bank today, her system must be able to process electronic orders. And it should be fast (Alice's bank is very successful and has many clients).

At least two decades ago, to increase the system throughput, Alice could have only needed to buy newer hardware, and her program would run faster. But 15–20 years ago, this *free lunch* ended [1]: Alice now has to use the computational power of several *hardware threads* to get the most of modern processors.

This is where a transactional memory can come in handy, and where *you* step in to help Alice.

Alice's program is currently written to run on a *single thread*. When Alice runs her code on her single-core processor (no concurrency), it runs as expected.

For instance, let's say that two clients, Bob and Charlie, wants to transfer money: both Bob and Charlie wants to wire each other (and they did not agree beforehand on "who owes who" to make only *one* transfer...). The pseudo-code for the "money transfer" function could be the following:

```
shared accounts as integer[];

fn transfer(src, dst, amount) {
    if (accounts[src] < amount) // Not enough funds
        return;                 // => no transfer
    accounts[dst] = accounts[dst] + amount;
    accounts[src] = accounts[src] - amount;
}
```

Bob wants to send 10 CHF to Charlie, and Charlie 15 CHF to Bob. Since there is only one hardware thread, each transfer will naturally be processed **one after the other**. That is, if Bob's request was received just before Charlie's request:

Bob Charlie	12 CHF 10 CHF	$\xrightarrow[\text{Bob's request}]{}$	Bob Charlie	2 CHF 20 CHF	$\xrightarrow[\text{Charlie's request}]{}$	Bob Charlie	17 CHF 5 CHF

Another possible execution, if instead Charlie's request was processed first by Alice's banking system:

Bob Charlie	12 CHF 10 CHF	$\xrightarrow[\text{Charlie's request}]{}$	Bob Charlie	12 CHF 10 CHF	$\xrightarrow[\text{Bob's request}]{}$	Bob Charlie	2 CHF 20 CHF

Even if Bob may not be very happy with this second possible outcome, it is still a *consistent* execution as far as the specification of the bank is concerned: when the funds are insufficient, no transfer happens.

Now, Alice wants to try her program (by assumption correct on a single-core processor) on brand-new, multi-core hardware. This way her system will be able to process transfers concurrently, (hopefully) answering faster her customers. So Alice, for testing purposes, runs the same program on multiple threads, each taking customer requests as they come. To analyze what outcomes are possible when two threads process the two respective transfers from Bob and Charlie in concurrence, we will rewrite the pseudo code to highlight each *shared* memory access, and in which order they are written in the program:

```
fn transfer(src, dst, amount) {
    let v0 = accounts[src];      // 0th read
    if (v0 < amount) // Not enough funds
        return;      // => no transfer
    let v1 = accounts[dst];      // 1st read
    accounts[dst] = v1 + amount; // 1st write
    let v2 = accounts[src];      // 2nd read
    accounts[src] = v2 - amount; // 2nd write
}
```

We assume that this pseudo-code runs on a pseudo-machine with *atomic* register[1], and that this pseudo-machine does not reorder memory accesses[2]: memory is always accessed in the same order as written in the program, and there exist a *total order* following which each atomic memory access happens. Supposing the account of Bob is at index `0` in `accounts`, and the account of Charlie at index `1`, a possible, concurrent execution is then (for clarity, `v0`, `v1`, `v2` have been replaced by `accounts[...]`):

Bob request's thread	**Charlie request's thread**
0th read `12 == accounts[0]`	
1st read `10 == accounts[1]`	
1st write `accounts[1] = 20`	
	0th read `20 == accounts[1]`
	1st read `12 == accounts[0]`
2nd read `12 == accounts[0]`	
2nd write `accounts[0] = 2`	
	1st write `accounts[0] = 27`
	2nd read `20 == accounts[1]`
	2nd write `accounts[1] = 5`

Figure 1: A concurrent execution that leads to an inconsistent state.

Under this concurrent execution, starting from Bob and Charlie having respectively $12 + 10 = 22$ CHF in total on their bank accounts, they end up with a total of $27 + 5 = 32$ CHF. Oups! Alice's system involutarily "offered" 10 CHF to Bob, by letting the thread from Charlie's request overwrites (in this specific execution) the deduction of 10 CHF made by the thread concurrently handling Bob's request. From the standpoint of the specification of the bank, this is an *inconsistency*.

A possible immediate solution could be to use a *mutual exclusion mechanism*, i.e. a *lock*:

```
fn transfer(src, dst, amount) {
    lock accounts { /* Only one thread at a time can enter this block:
        ...concurrent threads wait until no thread is in this block */
        let v0 = accounts[src];      // 0th read
        if (v0 < amount) // Not enough funds
            return;      // => no transfer
        let v1 = accounts[dst];      // 1st read
        accounts[dst] = v1 + amount; // 1st write
        let v2 = accounts[src];      // 2nd read
        accounts[src] = v2 - amount; // 2nd write
    }
}
```

Such a mechanism would prevent the execution from Figure 1, as only one thread accessing the *shared* variable `accounts` can run at any given time. But this mechanism also prevents multiple threads from processing transfers concurrently. Alice wanted to use multiple cores specifically to process multiple requests at the same time, answering faster her (many) customers. Always taking a single, coarse lock when accessing the shared memory is thus not a satisfying solution.

Actually, Alice noticed that some transfers can always run concurrently, without the need for *any synchronization*. For instance, if Bob wires Charlie and Dan wires Erin at the same time, no ordering of memory accesses can lead to an inconsistency. This is simply because the two sets of shared memory segments respectively accessed by these two transfers, i.e. {`accounts[0]`, `accounts[1]`} and {`accounts[2]`, `accounts[3]`}, do not overlap[3]: they do not *conflict*.

This is where the transactional memory is useful: it runs in parallel *transactions* that do not conflict, and synchronizes these which do conflict, (hopefully) allowing to make the most out of multi-core processors with minimal coding efforts. To get a glimpse of the general ideas behind the interface of the software transactional memory proposed in this project[4], let's rewrite (again) Alice's pseudo-code:

[1] The notion of *atomic register* is taught at the beginning of the course.
[2] The memory consistency models of modern hardware is outside the scope of this course.
[3] Also, memory segments that are *only read* by each thread can overlap without synchronization.
[4] Please refer to Section 2.2 for a precise definition of the interface of the library you have to implement.

```
// Initialization of the transactional memory, which holds the accounts
shared tm = new TransactionalMemory(/* ... */);
/* ...Initialize the array of accounts,
   at the "start address" of the shared memory... */

fn transfer(src, dst, amount) {
    let accounts = tm.get_start_address() as integer[];
    while (true) {
        try {
            let tx = tm.begin();
            if (tx.read(&accounts[src]) < amount) {
                tx.commit();
                break;
            }
            tx.write(&accounts[dst], tx.read(&accounts[dst]) + amount);
            tx.write(&accounts[src], tx.read(&accounts[src]) - amount);
            tx.commit();
            break;
        } except (RetryTransaction) {
            continue;
        }
    }
}
```

A transaction is *enclosed*, between calls to `tm.begin` and `tx.end`. Unlike with a lock, these functions (especially `begin`) may not block, allowing for concurrent executions.

Reads and writes (and *allocs* and *frees*) onto the shared memory are all controlled by the software transactional memory library: Alice's code is not allowed to e.g. directly read `accounts[dst]`. Instead, a call to `tx.read` must be made (`tx.write`, `tx.alloc`, `tx.free` for other operations). Notably, each of these operations can *abort*, requesting the caller to try again the same transaction.

1.2 Specification

The software transactional memory library sees and controls every access to the shared memory. The library can delay execution, return two different values when reading the same address from two different, concurrent transactions, etc. Basically, the goal of the library is to make *concurrent* transactions *appear as if* they were executed serially, without concurrency. (In the course, this notion is called *opacity*.) Also, once a transaction successfully committed, its modifications must be visible in subsequent transactions. A software transactional memory library that achieves this goal will be deemed correct.

Think of the serial executions of Bob's and Charlie's concurrent transfers (in Section 1.1): either the request from Bob was processed first, not seeing any modification from Charlie's request, or the opposite. They executed one after the other. Also, if Bob makes a new transaction (e.g. requesting its account details) *after* these two transactions committed, Bob must see both of their outcomes on his screen.

We can define three notions which, taken together, would satisfy the (informal) goal described above. These notions are: **snapshot isolation**, **atomicity**, and **consistency**. Namely:

- **Snapshot isolation** ensures that: (1) no write/alloc/free[5] in any non-committed transaction can be read/become available/unavailable in any other transaction, (2) all the reads/writes/allocs/frees made in a transaction *appear* to be made from the same *atomic snapshot* of the whole shared memory, excepted for (3) a transaction observes its own modifications (i.e. writes/allocs/frees).

  Example: if snapshot isolation had been satisfied in Figure 1, as Charlie's transaction already read `v1 == 12`*, then the* 2^{nd} *read from Charlie's thread could not have read* `v2 == 20`*, which either: (1) is the effect of a write in a non-committed transaction, if Bob's transaction had not committed, or (2) belongs to a different snapshot, if Bob's transaction had committed between the two reads.*

- **Atomicity** ensures that all the memory writes/allocs/frees of any committed transaction seem to have all happened at one indivisible point in time.

  Example: without atomicity, Bob's transaction could successfully commit, and another transaction (e.g. Charlie's transaction) could take a snapshot where only one of the two writes had been made.

[5] When Alice's program calls `tx.free`, the transactional memory library is not required to immediately call libc's `free`.

- **Consistency** ensures, for a committing transaction T, that: (1) none of the transaction that committed since the atomic snapshot of T (a) freed a segment of memory read or written by T or (b) allocated a segment that overlaps with a segment allocated by T, and (2) each read made by T in its snapshot would read the same value in a snapshot taken after the *last*[6] committed transaction.

  Example: let's consider again Figure 1. Both Bob's and Charlie's transactions run concurrently with (in this example) the same snapshot: Bob has 12 CHF and Charlie 10 CHF. It is consistent for Bob's transaction to commit first, as there has been no committed transaction since Bob's transaction begun and thus the shared memory remained the same since the snapshot of Bob's transaction. Regarding Charlie's committing transaction, which read memory locations that were modified since its atomic snapshot, it would not be consistent to commit: Charlie's transaction must be ***retried****. Now if we consider the other example of Bob wiring Charlie while Dan wires Erin, it would be consistent for both transactions to commit, since they accessed different words of the shared memory.*

Side notes:

- There are several definitions of *consistency* in the literature. Here we provide a sensible definition for this project, while being arguably more precise/specific/actionable than some other existing definitions (e.g. *"Data is in a consistent state when a transaction starts and when it ends."* [3]).
- Point (3) of *snapshot isolation* is usually found in *consistency*.

1.3 Possible implementations (in English)

This section describes the reference implementation and a possible transactional memory for this project. You are free to implement *any* other software transactional memory (see the rules in Section 2.1.3). More ideas can be found while studying:

- TinySTM: `https://github.com/patrickmarlier/tinystm`
- LibLTX: `https://sourceforge.net/projects/libltx`
- stmmap: `https://github.com/skaphan/stmmap`

1.3.1 Using a coarse-lock (i.e. the "reference" implementation)

This implementation is very simple to describe: the transactional memory library uses one single mutex to (trivially) serialize transactions made onto the shared memory.

When a transaction begins (i.e. `tm.begin` is called), the single mutex is taken. When the transaction ends, it always commits (i.e. never retries), and the single mutex is released. Read, write, allocation and deallocation operations are directly mapped to reading/writing the memory at the requested addresses, and essentially calling libc's `malloc`/`free`[7].

This approach is obviously correct, as it directly consists in both (1) executing transactions serially and (2) ensuring new transactions sees all the writes/allocs/frees from the last committed transaction.

From the prism of the three notions laid above, *snapshot isolation* is guaranteed because no (concurrent) transaction can run to observe the memory effects of the pending transaction holding the lock, until the pending transaction commits, and so, releases the lock. *Atomicity* is directly guaranteed by the lock, and ditto for *consistency*, since there cannot be two transactions running concurrently: the pending transaction is the only one able to alter the state of the shared memory.

1.3.2 Dual-versioned transactional memory

This is not the fastest possible transactional memory for the grading workload, nor the simplest you can implement[8]*. It is only provided as a suggestion, if you are not willing to look for a possibly more efficient (or less complex) implementation. Also, only the high-level concepts and logic will be presented. Low-level details and optimizations, e.g. actual memory layouts, are omitted on purpose, and for you to find.*

[6] This notion of *last* committed transaction implicitly relies on the existence of a *total order* following which transactions commit. To further clarify this point: if T_1 and T_2 are the only two transactions committing concurrently (and both can commit), then either T_1 will be the last committed transaction for T_2, or T_2 will be the last committed transaction for T_1.

[7] The implementation also needs to keep track of the allocated segments, to prevent any memory leak: see Section 2.2.

[8] Any reasonably optimized implementation will nevertheless be faster than the reference.

Through the use of a very basic multiversionning scheme, this implementation aims at making sure read-only transactions can both: run concurrently with read-write transactions, and never fail. Two copies of every (aligned) *word* of the size of the requested alignment (see Section 2.2.2) are kept. When a transaction reads/writes words in shared memory, the transactional library has to decide for each word which of the two copies must be read/written, or if none can be read/written and the transaction must abort.

One key structure that can considerably simplify the design (at the expense of performance though) is what we will call the *Thread Batcher* (or more simply, the *Batcher*). The goal of the Batcher is to *artificially* create points in time in which no transaction is running. This goal might sound uncanny, but it is not: a mere mutex, as the one used in the reference implementation, creates such points in time. Indeed, when the transaction that got the global lock releases it, no transaction is running.

You can think of the Batcher as a special kind of mutex: while a mutex lets *only one* blocked thread enter and leave no matter how many threads were blocked waiting, the Batcher lets *each and every* blocked thread(s) enter together when the last thread (from the previous batch) leaves. So the interface of the Batcher is pretty much the same as the one of a mutex: `enter` and `leave` (analogous to `lock` and `unlock`), and one special function called `get_epoch` that can only be called by a thread after it entered (and before it leaved) the Batcher. `get_epoch` returns an integer that is unique to each batch of threads (e.g. a counter, incrementing when the last thread of each batch leaves, would do well).

Batcher pseudo-code, assuming all functions execute in one indivisible point in time (they are *atomic*):

```
Data: counter: integer, remaining: integer, blocked: list of threads
fn get_epoch()
    return counter;
end
fn enter()
    if remaining = 0 then
        remaining = 1;
    else
        append current thread to blocked;
        wait until "woken up";
    end
end
fn leave()
    decrement remaining by 1;
    if remaining = 0 then
        increment counter by 1;
        remaining = length of blocked;
        "wake up" every thread in blocked;
        empty list blocked;
    end
end
```

Internally with this implementation, each shared memory region holds one Batcher instance. When a transaction begins on a shared memory region, `enter` must be called. And the last operation made from inside a transaction, when it ends (no matter if the transaction was successful), should be to call `leave`.

At this point, we have "batched" transactions running onto the shared memory region. We will call an "epoch" the time span during which each batch of transactions runs.

With our dual-versioned implementation, each segment of n word(s) (the size of a word here is the requested alignment) is duplicated, and for each word there is a *control* structure to decide which version to read/write, or if the transaction must abort. A schematic representation of one segment is given below:

word 0	word 1	word 2	word 3
control	control	control	control
copy A	copy A	copy A	copy A
copy B	copy B	copy B	copy B

. . .

For the sake of the presentation laid below, the word indexes above (i.e 0, 1, . . .) will be assumed unique *for the whole shared memory region: there is no two words in the same region that have the same index.*

The control structure contains the following pseudo-fields[9]:

- Which copy is "valid" from the previous epoch (i.e. either A or B).
  This copy will be called the "readable copy", and the other copy will be called the "writable copy".
- The "access set" of read-write transaction(s) which have accessed the word in the current epoch.
  Do not implement an actual set in any (optimized) implementation: this set will only be used to tell whether a transaction can write to the word. Namely, if at least one ***other*** *transaction has accessed (i.e. read or written) this word in the same epoch, the write cannot happen.*
- Whether the word has been written in the current epoch.

The pseudo-code of the read/write/alloc/free/commit operations (+ 2 helper functions) can then be:

```
fn read_word(index, target)
    if the transaction is read-only then
        read the readable copy into target;
        return the transaction can continue;
    else
        if the word has been written in the current epoch then
            if the transaction is already in the "access set" then
                read the writable copy into target;
                return the transaction can continue;
            else
                return the transaction must abort;
            end
        else
            read the readable copy into target;
            add the transaction into the "access set" (if not already in);
            return the transaction can continue;
        end
    end
end

fn write_word(source, index)
    if the word has been written in the current epoch then
        if the transaction is already in the "access set" then
            write the content at source into the writable copy;
            return the transaction can continue;
        else
            return the transaction must abort;
        end
    else
        if at least one other transaction is in the "access set" then
            return the transaction must abort;
        else
            write the content at source into the writable copy;
            add the transaction into the "access set" (if not already in);
            mark that the word has been written in the current epoch;
            return the transaction can continue;
        end
    end
end
```

[9] Your implementation may not declare a control structure with this exact list of fields, each field with the exact same semantic: your implementation should probably have more fields, with adjusted semantic, for the purpose of optimization.

```
fn read(source, size, target)
    foreach word index within [source, source + size[ do
        result = read_word(word index, target + offset);
        if result = transaction must abort then
            return the transaction must abort;
        end
    end
    return the transaction can continue;
end

fn write(source, size, target)
    foreach word index within [target, target + size[ do
        result = write_word(source + offset, word index);
        if result = transaction must abort then
            return the transaction must abort;
        end
    end
    return the transaction can continue;
end

fn alloc(size, target)
    allocate enough space for a segment of size size;
    if the allocation failed then
        return the allocated failed;
    end
    initialize the control structure(s) (one per word) in the segment;
    initialize each copy of each word in the segment to zeroes;
    register the segment in the set of allocated segments;
    return the transaction can continue;
end

fn free(target)
    mark target for deregistering (from the set of allocated segments)
     and freeing once the last transaction of the current epoch leaves
     the Batcher, if the calling transaction ends up being committed;
    return the transaction can continue;
end

fn commit()
    foreach written word index do
        defer the swap, of which copy for word index is the "valid"
         copy, just after the last transaction from the current epoch
         leaves the Batcher and before the next batch starts running;
    end
    return the transaction has committed;
end
```

As for the Batcher pseudo-code, we assume the functions outlined above all execute *atomically*. Numerous implementation details, e.g. how to access the control structure of each word, how to keep track of the allocated segments, etc, are intentionally left completely unspecified.

For reference: (optimized) implementations can reach speedups above $\times 2.5$ against the *grading* workload, and the implementation of the TA is less than 1000 lines long (all comments and spaces included).

2 The project

You are expected to be reasonably fluent in C11 or C++17 for this project. A (very brief) introduction to the memory model of C11/C++11, along with available atomic operations, are provided in Section 3.

2.1 Practical information

This project is associated with a git repository, hosted on GitHub. It will be available at `https://github.com/LPD-EPFL/CS453-2020-project` when the semester starts. Its layout is the following:

`grading/` Directory containing the source code of the evaluation program, to test your solution on your own machine. The evaluation server (see below) runs the exact same binary, possibly with a different seed. Section 2.1.1 further details how to test and later submit your code.

`include/` C11 and C++17 header files that define the public interface of the STM.

`template/` Template (in C11) of your implementation, but you are free to replace everything and write C++17 instead. See section 2.1.3 for more information.

`reference/` Reference, single lock-based implementation (see Section 1.3.1).
Note that you are not allowed to write an implementation that is equivalent to this reference implementation. See section 2.1.3 for more information.

The specification of the evaluation server, on which your submission will be run, are the following:

CPU	2 (dual-socket) Intel(R) Xeon(R) 10-core CPU E5-2680 v2 at 2.80GHz ($\times$2 hyperthreading $\Rightarrow$ 40 virtual cores)
RAM	256 GB
OS	GNU/Linux (distribution: Ubuntu 18.04 LTS)

2.1.1 Testing and submission

At this point, you should have received your *unique (and secret) user identifier* by mail.
If not, ask the TAs as soon as possible: without this identifier, you cannot submit your implementation.

The proposed workflow is the following:

1. Clone/download the repository.
2. Make a local copy of/rename the template directory with your SCIPER number:
   ```
   you@your-pc:/path/to/repository$ cp -r template 123456
   ```
   Be aware `make` is sensitive to spaces in the path/names.
3. Complete or completely rewrite (your copy of) the template with your own code.
   (a) Complete/rewrite your code; only the interface should be kept identical.
   (b) Compile and test *locally* with:
   ```
   you@your-pc:/path/to/repository/grading$ make build-libs run
   ```
   (c) Send your code to the evaluation server for testing on the evaluation server; see below.
   (d) Repeat any of the previous steps until you are satisfied with correctness and performance.

The procedure to send your code for evaluation is the following:

1. Zip your modified (copy of the) template directory (not directly its content).
2. Send your code for evaluation with the `submit.py` script:
   ```
   you@your-pc:/path/to/repository$ python3 submit.py --uuid <...> 123456.zip
   ```
   The UUID (Unique User IDentifier) to use is the one you should have received by mail.

Your request will be queued on the evaluation server, and you will receive the results of both the compilation and the execution of your implementation (including its speedup vs. the reference one) as soon as it finishes running. You can submit code as often as you want until the deadline.

Only the (correct) submission with the highest speedup will be stored persistently and considered for grading, so no need to resubmit your best implementation just before the deadline. The TAs may still ask you to provide a copy of your best implementation though (possibly after the deadline), so do not delete anything before the end of the semester (2021-01-30).

As an option from the `submit.py` script, you can *force-replace* your submission with another one. In this case no compilation nor performance check is performed. The submission script also allows you to download the submission considered for your grade, so you can make sure the system has the version you want it be considered for grading. You can display the command-line help for this tool with:

```
you@your-pc:/path/to/repository$ python3 submit.py -h
```

Also note that the automated submission system has no false negative, i.e. a correct implementation will always be deemed correct; unless it is more than 8 times slower than the reference (see Section 2.1.2).

On the other hand, the system has false positive (i.e. incorrect implementations deemed correct). Unless your submission includes portions of code that appear to be specifically designed to *trick* the submission system into considering an incorrect implementation correct, if the submission system qualifies your implementation as correct, it will be graded as a correct one.

2.1.2 Grading

Correctness is a must

The executed transactions must satisfy the specification (sections 1.2 and 2.2).
No correctness ⇒ no "passing" grade for the project (i.e. grade < 4).

Performance is required to get the maximum grade

The only considered metric is the *number of transactions committed per second*.

Providing a correct implementation guarantees you at least a grade of 4 for the project, although an implementation that is at least 8 times slower than the reference (coarse lock-based) implementation will not be deemed correct, regardless of whether a longer execution time would have allowed it to finish. Please also note: an implementation that leaks memory will have its grade capped at 5.

Given its speedup relative to the reference, the grade of a correct implementation (that does not leak) is:

Speedup s	**Grade** g
$s < 1/8$	$g < 4$
$s \in [1/8, 1]$	$g = 4 + \frac{8s-1}{7}$
$s \in [1, 4]$	$g = 5 + \frac{s-1}{3}$
$s > 4$	$g = 6$

For reference, the maximum speedup obtained last year on this project is: ×15.04. The associated implementation consisted in about ∼600 lines of C++17 (including comments).

2.1.3 Other rules

The deliverable is to be written either in C11 or in C++17. The provided template code is written in C11, and you can overwrite it completely if you deem it necessary. In case you want to develop in C++17 instead of C11, you should use the associated `#include <tm.hpp>` instead of `#include <tm.h>`.

You can use any external library, as long as they do not solve a concurrent programming problem. One exception to this rule is if the external library you want to use only solves *elementary* concurrent programming problems, e.g. the library implements mutexes and condition variables. If you are in doubt whether you can use a specific library, please ask the TAs. Please keep in mind a few limitations though:

- The size of each submission is limited to 100 kB (both compressed and uncompressed), so you cannot only include in your submission libraries that are limited in size.
- The compilation time is limited to 10 s.

Note that you are not allowed to write an implementation that is equivalent to the reference implementation, i.e., that uses a single, global lock to serialize transactions; even if this lock is a reader-writer lock instead of a classic mutex. When in doubt, ask the TAs right away for clarifications.

You may read and take inspiration from existing STM libraries (see Section 1.3), and collaborate with other students (e.g. share ideas, help debug each others' code, etc). But at the end of the day it must be *your own code* that carries out transactions in your submission.

2.2 Interface of the software transactional memory

To use this Software Transactional Memory (STM) library, the *user* of the library (i.e. Alice, or more specifically here, the *grading* tool) first creates a new shared memory region. A shared memory region is a non-empty set of non-overlapping shared memory segments. Shared memory region creation and destruction are respectively managed by `tm_create` and `tm_destroy`. A shared memory region is always created with one first, non-deallocable shared memory segment. Additional shared memory segments (de)allocation are managed by `tm_alloc` and `tm_free`. The content of a shared memory regions can only be accessed from inside a transaction, and solely by using the functions described in Section 2.2.2.

A transaction consists of a sequence of `tm_read`, `tm_write`, `tm_alloc`, `tm_free` operations in a shared memory region, enclosed between a call to `tm_begin` followed later by a call to `tm_end`. A transaction cannot access more than one shared memory region[10]. A transaction either commits its speculative updates to the shared memory region when `tm_end` is reached, or aborts and discards its speculative updates (see Section 2.2.2). When a transaction is aborted, the user (i.e. the *grading* tool in this project) is responsible for retrying the same transaction (i.e. going back to the same `tm_begin` call site).

2.2.1 Data structures

The interface of the transactional memory library is composed of three data structures:

- `shared_t`: an opaque handle representing a shared memory region
- `tx_t`: an opaque handle representing a pending transaction on a shared memory region
- `void*`: a pointer to the first word of a shared memory segment

Shared memory segments are (de)allocated like with e.g. `malloc`/`free`: `tm_alloc` returns the address of the first word, which is also the address to pass to `tm_free` to free the segment. Words on the segment are addressed as with C/C++ arrays, by adding an offset to the address of the first word of the segment. (And of course, the user will call `tm_read`/`tm_write` instead of directly performing reads/writes.)

Note that, since the layout of the structure pointed to by `shared_t`, `tx_t`, and `void*` (as a pointer to a word on a shared memory segment) are all opaque to the user, you are free to design their internals the way you want. Feel free to take some inspiration from how the reference implemented these types.

2.2.2 Functions to implement

```
shared_t tm_create(size_t size, size_t align)
```

Create (i.e. allocate + init) a new shared memory region, with one first allocated segment of the requested size and alignment.

Parameter	Description
`size`	Size of the first allocated segment of memory (in bytes), must be a positive multiple of the alignment
`align`	Alignment (in bytes, must be a power of 2) that the shared memory region must provide, and that each memory access made on this shared memory region will have to satisfy

Returns: Opaque shared memory region handle, `invalid_shared` on failure.

Additional comments:

- The requested alignment in that function will be the alignment assumed in every subsequent memory operation.
- The first allocated segment must be initialized with zeroes.
- The first allocated segment cannot be freed with `tm_free`.
- This function can be called concurrently.

[10] You may notice that the *grading* tool will actually create only one memory region. But your library should be able to handle several distinct regions, and thus not use e.g. global variables to track transactional states.

```
void tm_destroy(shared_t shared)
```

Destroy (i.e. clean-up + free) a given shared memory region.

Parameter	Description
`shared`	Handle of the shared memory region to destroy

Additional comments:

- No concurrent call for the same shared memory region.
- It is guaranteed that when this function is called the associated shared memory region has not been destroyed yet.
- It is guaranteed that no transaction is running on the shared memory region when this function is called.
- The first allocated segment, and all the segments that were allocated with `tm_alloc` but not freed with `tm_free` at the time of the call, must be freed by this function.

```
void* tm_start(shared_t shared)
```

Get a pointer to the first allocated segment of the shared memory region.

Parameter	Description
`shared`	Handle of the shared memory region to query

Returns: Pointer to the first word of the first allocated segment.

Additional comments:

- This function can be called concurrently.
- The returned address must be aligned on the shared region alignment.
- This function never fails: it must always return the address of the first allocated segment, which is not free-able.
- The returned pointer must not be `NULL` (or `nullptr` in C++), and must not change between invocations.

```
size_t tm_size(shared_t shared)
```

Get the size to the first allocated segment of the shared memory region.

Parameter	Description
`shared`	Handle of the shared memory region to query

Returns: Size (in bytes) of the first allocated segment.

Additional comments:

- This function can be called concurrently.
- The returned size must be a multiple of the shared region alignment.
- This function never fails: it must always return the size of the first allocated segment, which is not free-able.
- The size of the first allocated segment is a constant, set with `tm_create`.

```
size_t tm_align(shared_t shared)
```

Get the required alignment for memory accesses on the shared memory region.

Parameter	Description
`shared`	Handle of the shared memory region to query

Returns: Alignment used for this shared memory region (in bytes).

Additional comments:

- This function can be called concurrently.
- The alignment of the shared memory region is a constant, set with `tm_create`.

```
tx_t tm_begin(shared_t shared, bool is_ro)
```

Begin a new transaction on the given shared memory region.

Parameter	Description
`shared`	Shared memory region to begin a transaction on
`is_ro`	Whether the transaction will only perform read(s)

Returns: Opaque transaction handle, `invalid_tx` on failure.

Additional comments:

- This function can be called concurrently.
- There is no concept of nested transactions, i.e. one transaction begun in another transaction.
- If `is_ro` is set to true, then only `tm_read` will be called from this transaction.

```
bool tm_end(shared_t shared, tx_t tx)
```

End the given transaction.

Parameter	Description
`shared`	Shared memory region associated with the transaction
`tx`	Transaction to end

Returns: `true`: the whole transaction committed, `false`: the transaction must be retried

Additional comments:

- This function can be called concurrently, but concurrent calls must be made with at least a different `shared` parameter or a different `tx` parameter.
- This function will not be called by the user (e.g. the *grading* tool) if any of `tm_read`, `tm_write`, `tm_alloc`, `tm_free` already notified that `tx` was aborted.

```
bool tm_read(shared_t shared, tx_t tx, void const* source, size_t size, void* target)
```

Read operation in the transaction, source in the shared region and target in a private region.

Parameter	Description
`shared`	Shared memory region associated with the transaction
`tx`	Transaction to use
`source`	Source (aligned) start address (in shared memory)
`size`	Length to copy (in bytes)
`target`	Target (aligned) start address (in private memory)

Returns: `true`: the transaction can continue, `false`: the transaction has aborted

Additional comments:

- This function can be called concurrently, but concurrent calls must be made with at least a different `shared` parameter or a different `tx` parameter.
- The private buffer `target` can only be dereferenced for the duration of the call.
- The length `size` must be a positive multiple of the shared memory region's alignment, otherwise the behavior is undefined.
- The length of the buffers source and target must be at least `size`, otherwise the behavior is undefined.
- The `source` and `target` addresses must be a positive multiple of the shared memory region's alignment, otherwise the behavior is undefined.

```
bool tm_write(shared_t shared, tx_t tx, void const* source, size_t size, void* target)
```

Write operation in the transaction, source in a private region and target in the shared region.

Parameter	Description
`shared`	Shared memory region associated with the transaction
`tx`	Transaction to use
`source`	Source (aligned) start address (in private memory)
`size`	Length to copy (in bytes)
`target`	Target (aligned) start address (in shared memory)

Returns: `true`: the transaction can continue, `false`: the transaction has aborted

Additional comments:

- This function can be called concurrently, but concurrent calls must be made with at least a different `shared` parameter or a different `tx` parameter.
- The private buffer `source` can only be dereferenced for the duration of the call.
- The length `size` must be a positive multiple of the shared memory region's alignment, otherwise the behavior is undefined.
- The length of the buffers source and target must be at least `size`, otherwise the behavior is undefined.
- The `source` and `target` addresses must be a positive multiple of the shared memory region's alignment, otherwise the behavior is undefined.

`alloc_t tm_alloc(shared_t shared, tx_t tx, size_t size, void** target)`

Shared memory segment allocation in the transaction.

Parameter	Description
`shared`	Shared memory region associated with the transaction
`tx`	Transaction to use
`size`	Alloction requested size (in bytes)
`target`	Pointer in private memory receiving the address of the first word of the newly allocated, aligned segment

Returns: `success_alloc`: the allocation was successful and transaction can continue,
`abort_alloc`: the transaction has aborted,
`nomem_alloc`: the memory allocation failed (e.g. not enough memory)

Additional comments:

- This function can be called concurrently, but concurrent calls must be made with at least a different `shared` parameter or a different `tx` parameter.
- The pointer `target` can only be dereferenced for the duration of the call.
- The value of `*target` is defined only if `success_alloc` was returned.
- The value of `*target` after the call if `success_alloc` was returned must not be `NULL` (or `nullptr` in C++).
- When `nomem_alloc` is returned, the transaction is not aborted.
- The allocated segment must be initialized with zeroes.
- Only `tm_free` can be used to free the allocated segment.
- The length `size` must be a positive multiple of the shared memory region's alignment, otherwise the behavior is undefined.

`bool tm_free(shared_t shared, tx_t tx, void* target)`

Shared memory segment deallocation in the transaction.

Parameter	Description
`shared`	Shared memory region associated with the transaction
`tx`	Transaction to use
`target`	Pointer to the first word of the allocated segment to deallocate

Returns: `true`: the transaction can continue, `false`: the transaction has aborted

Additional comments:

- This function can be called concurrently, but concurrent calls must be made with at least a different `shared` parameter or a different `tx` parameter.
- This function must not be called with `target` as the first allocated segment (the address returned by `tm_start`).

3 Concurrent programming with shared memory

Reference (C/C++): `https://en.cppreference.com/w/`

- Order of evaluation: `https://en.cppreference.com/w/cpp/language/eval_order`
- Undefined behavior: `https://en.cppreference.com/w/cpp/language/ub`
- Atomic types in C11: `https://en.cppreference.com/w/c/atomic`
- Atomic types since C++11: `https://en.cppreference.com/w/cpp/atomic`
- Memory model: `https://en.cppreference.com/w/cpp/language/memory_model`
- Memory order: `https://en.cppreference.com/w/cpp/atomic/memory_order`

It is very much recommended to at least skim through these references, especially if you had never heard before about *undefined behavior* and the *sequenced-before* rules. These are two C/C++ notions that are assumed to be known by the student, as these notions are not from concurrent programming.

3.1 Atomic operations since C11/C++11

In C and C++, it is a *data race* when a thread writes to a memory location concurrently read of written by another thread, unless both accesses are *atomic* operations (see below) or one of the conflicting accesses *happens-before* the other (see Section 3.2). If a data race occurs, the behavior of the program is undefined.

In C11, with `#include <stdatomic.h>`, the atomic version of a type `T` is declared with `_Atomic T`.
Since C++11, with `#include <atomic>`, it can be declared with `::std::atomic<T>`, with `T` being *TriviallyCopyable*, *CopyConstructible* and *CopyAssignable*.

In both C11 and C++11 (and above), the operators for these types are overloaded and use atomic counterparts for these operations. For instance, the increment operator `++x` on an atomic object `x` is equivalent to calling `atomic_fetch_add(&x, 1)` in C, and `x.fetch_add(1)` in C++. As a quick overview:

Common name	C function(s)	C++ method(s)
Read	`atomic_load`	`.load`
Write	`atomic_store`	`.store`
Fetch-and-...	`atomic_fetch_...`	`.fetch_...`
Swap	`atomic_exchange`	`.exchange`
Compare-and-Swap	`atomic_compare_exchange_weak`	`.compare_exchange_weak`
	`atomic_compare_exchange_strong`	`.compare_exchange_strong`

(The only difference between the *weak* and *strong* version of a compare-and-swap is that the *weak* version may (spuriously) act as if the stored value was not the expected value, even if they were equal.)

Each of these atomic operations either have: counterparts which names end in `_explicit` and take one (or two) more parameter(s) (in C11), or defaulted parameters (since C++11). These parameters specify *ordering constraints* when the atomic operation is used to *synchronize* with concurrent threads. Understanding these notions, and using the ordering parameters, are *optional* for the project. *Memory ordering* is introduced in Section 3.2 only for completeness, and for students interested to know more.

Real uses of these atomic operations are available in the provided repository, in `class Sync` in file *grading/grading.cpp*, and in the two custom definitions of `struct lock_t` in file *reference/tm.c*.

3.2 Memory ordering in C11/C++11: a quick overview

Understanding this subsection (which is actually just a quick overview) is optional for this project. But if you are interested on this matter, this section is not enough so here are a few pointers to start with:

- Introduction to lock-free programming
  `https://preshing.com/20120612/an-introduction-to-lock-free-programming/`
- Memory Barriers Are Like Source Control Operations
  `https://preshing.com/20120710/memory-barriers-are-like-source-control-operations/`
- Acquire and Release Fences Don't Work the Way You'd Expect
  `https://preshing.com/20131125/acquire-and-release-fences-dont-work-the-way-youd-expect/`

Let's consider a basic, *single-threaded* C11/C++11 code snippet:

```
int a = 0;
int b = 0;
printf("a = %d, b = %d\n", a, b);
a = 1;
printf("a = %d, b = %d\n", a, b);
b = 1;
printf("a = %d, b = %d\n", a, b);
```

Following the *sequenced-before* rule (which is very intuitive in this example), we expect to see printed:

```
a = 0, b = 0
a = 1, b = 0
a = 1, b = 1
```

Indeed: it is not possible to read `a = 0, b = 1`, as `a = 1` is *sequenced-before* `b = 1`.

Now, let's split this code snippet, to run it on *two concurrent threads*:

```
// Global variables   // Thread 1   // Thread 2
int a = 0;            a = 1;       printf("a = %d, b = %d\n", a, b);
int b = 0;            b = 1;
```

Given that the two global variables are initialized before any of the two thread started running, which pair(s) of values (`a`, `b`) would the second thread be able to print? Answer: *any* of the following pairs[11]:

```
a = 0, b = 0
a = 1, b = 0
a = 1, b = 1
a = 0, b = 1
```

But why?

A technical reason is that both the compiler and the processor sometimes reorder memory operations. For instance in the given example, the compiler might see fit to issue the read instruction(s) for `b` before the one(s) for `a`, as the compiler is not constrained (not even by the *sequenced-before* rules) to read `a` before `b`. The rule of thumb for compiler reordering and optimizations in general might be: "as long as the result of a single-threaded execution is not altered, (almost) any change is allowed" (the *as-if* rule).

Hardware reordering can also happen very frequently, depending on the memory model provided by the architecture your program run on. More on that in specific courses, e.g. CS-471.

These reorderings can become a problem for multi-threaded, concurrent programming though. In C11 and since C++11, both hardware and compiler reorderings can be explicitly prevented (both at once) through the use of the *ordering parameter(s)* in atomic operations or *thread fences* (a.k.a. *barriers*).

Similar to the *sequenced-before* rule, C11 and C++11 introduce several new notions, among which *synchronize-with* and *happen-before*. For a formal description of the whole model, with examples, please see: `https://en.cppreference.com/w/cpp/atomic/memory_order#Formal_description`.

Informally, the keystone is that the modifications made to an atomic variable happen in a total order, on which every thread agree. (This is not to be confounded with a total order between modifications happening to all the atomic variables.) From this *agreement*, the remaining step is to prevent some operations (e.g. reads and/or writes) from being reordered before/after some important atomic operations.

As an example, let's consider the implementation of a lock. The interface is composed of two functions: `lock` and `unlock`. A lock protects some parts of the shared memory. So, in particular, we do not want the *lock-protected* reads to happen before the lock is taken. Ditto for the *lock-protected* writes: they must not happen after the lock is released, otherwise the next thread owning the lock could see an inconsistent view (not all writes visible by the time the lock is taken). So a correct lock implementation could be:

[11] Actually, this program has two obvious *data races*: its behavior is technically *undefined*, (any/no)thing could be printed.

```cpp
// Shared lock
using Mutex = ::std::atomic<bool>;
Mutex mutex = false;

void lock(Mutex& mutex) {
    bool expected = false;
    while (!mutex.compare_exchange_weak(expected, true,
                ::std::memory_order_acquire, ::std::memory_order_relaxed)) {
        expected = false;
        while (mutex.load(::std::memory_order_relaxed)) // Spin wait
            pause();
    }
}

void unlock(Mutex& mutex) {
    mutex.store(false, ::std::memory_order_release);
}
```

The use pattern for a thread would then be: (1) `lock`, (2) read/write shared memory, (3) `unlock`. The *acquire* semantic (third parameter of `compare_exchange_weak`, see the documentation for this method) ensures that a successful compare-and-swap *happens-before* read operations written after it. Similarly, the *release* semantic on the `store` ensures that all the writes operations written before *happen-before* the release. Finally when the lock is released then taken again (by the compare-and-swap), since there is a total order for all the operations on the `mutex`, the write operations of the previous lock holder *happen-before* any read operation from the new lock holder. The view is consistent.

One last note, regarding the use of *thread fences* instead of the *memory ordering parameters* of atomic operations. Thread fences (a.k.a. *barriers*) prevent the reordering of reads and writes before/after the fence. This is (almost) just a change of perspective from using memory ordering parameters: instead of preventing memory operations from being reordered across a specific atomic operation, the fence prevents some types of operations (e.g. reads, writes, or both) before/after the fence from being reorder after/before it. This is slightly more constraining than memory ordering parameter(s) on atomic operations.

3.3 From C11/C++11 to the model used in the course

This section will be very short. In C11 and since C++11, operations with *sequential consistency* semantic:

- globally follow a total order,
- *happen-before* each other in program order,

which is the semantic of the model introduced at the beginning of the course.

Using this semantic, we can correct the (trivial) example given at the beginning of Section 3.2:

```cpp
// Global variables          // Thread 1   // Thread 2
::std::atomic<int> a = 0;    a = 1;        int x = a;
::std::atomic<int> b = 0;    b = 1;        printf("a = %d, b = %d\n", x, b);
```

Since *sequential consistency* is the default ordering for atomic operations, the second thread will only be able to print the following pairs:

```
a = 0, b = 0
a = 1, b = 0
a = 1, b = 1
```

The pair `a = 0, b = 1` cannot be printed anymore, as with sequential semantic, the write `a = 1` *happens-before* `b = 1`. Note thought that the second thread first reads `a` into the local variable `x`, since in C/C++, the *order of evalutation* of parameters in a function call is undefined.

References

[1] Herb Sutter, *The Free Lunch Is Over, A Fundamental Turn Toward Concurrency in Software*, 2005, `http://www.gotw.ca/publications/concurrency-ddj.htm`

[2] Maurice Herlihy, Nir Shavit, The Art of Multiprocessor Programming, 2011

[3] IBM Knownledge Center, ACID properties of transactions, `https://www.ibm.com/support/knowledgecenter/en/SSGMCP_5.4.0/product-overview/acid.html`