Lab 2: Multiprocessors and Memory Models

2025-03-13

Lägg till kommentar

Information

Introduction

The purpose of this lab is to show the need for synchronization on modern multiprocessors and how this can be achieved in practice. In this module, we have seen how even the simplest operation requires multiple steps and coordination between different parts of the system. For example, something as trivial as incrementing an integer by one requires (at least) two memory operations that might need to traverse the whole memory hierarchy while also interacting with the other cores and processors present in the system. We have also seen how such interactions happen on the coherence level and how these are made visible to the users of the system as part of the memory model. To demonstrate this, the lab is split into two parts, A and B. In Part A we will see how synchronization primitives (such as locks and atomics) can be used, while in Part B we will delve deeper and implement our own locks, using the primitives we saw in Part A.

It is highly recommended that you solve this assignment in groups of two (or three) students, using the same groups as for Lab 1. Contact the teaching assistants if you have any problems with your current group. Much like Lab 1, the lab assignment will be examined during the two lab slots. During the examination, you will be asked to demonstrate and explain your solutions. Each group member should be able to answer questions for all parts of the lab. We might also ask you additional questions (relevant to the lab material) not found in this document. Other than understanding the code itself, it is important to also understand the synchronization patterns and which of the four orderings ("load -> load", "load -> store", "store -> load", "store -> store") need to be enforced for each part of the code. Also, **make sure to read this document carefully**, there is a lot of information to digest, and **refer to the linked documentation** for help with the various C++ types and functions we will be using in the lab.

LAB2 Prep is not mandatory. There are two lab slots, Lab 2A and Lab 2B, where examination will take place. You are only required to attend one of them. Please sign up for which slot you want to attend on **the calendar (https://uppsala.instructure.com/calendar?include_contexts=course_94980#view_name=month&view_start=2025-03-01)**. Remember that it is mandatory to pass the lab in order to pass the course, with the exception of the bonus questions, which give you one bonus point.

TA email: **it-avdark-ta@lists.uu.se (mailto:it-avdark-ta@lists.uu.se)**

Synchronization and Atomics

What constitutes "synchronization" depends on the system and the model we are talking about, but in our case we are concerned with communication between different threads executing on a multi-threaded processor. Specifically, we will focus on what synchronization is and how it can be achieved in the context of modern C++.

In the tasks (which can be found later) you will be first asked to use the synchronization methods found in modern C++ to implement an atomic counter class in different ways. Conceptually, an atomic counter is not much different from an integer variable that you can increment and decrement at will, with the difference that it is possible to do so from multiple threads simultaneously while keeping the result consistent. Here is a simple example of an atomic counter in action:

```
// Main thread
atomic_counter counter = 0;

// Thread 1                      // Thread 2
counter.increment();             counter.decrement();
counter.increment();             counter.decrement();

// Main thread
wait_for_threads_to_finish();
print(counter) // This will print 0;
```

It should be clear that if we were to simply replace the `atomic_counter` with a normal integer variable and no other synchronization then this example would not work (If it isn't clear, please rewatch the videos on Studium).

In Part A of the lab, you will be asked to implement such an atomic counter using various methods, including atomic primitives found in the language. We also provide a simple test program that performs the above operation (only many more times) so you can test your implementation. In Part B, after you have understood how synchronization is used and why it is important, you will instead be asked to implement your own synchronization methods, using other synchronization methods that exist in the language.

Modern C++

As explained above, what constitutes "synchronization" greatly depends on the context. For this lab, we will be focusing on the software level, specifically modern C++. Memory models are not exclusive to the caches or even the hardware, and many modern programming languages, Ses, etc, have their own memory models. Thankfully, in modern C++, we only need to be concerned with the memory model of the language, as the compiler makes sure to emit the appropriate code to match the hardware memory model to that of the language.

C++ Memory Model

Since the introduction of C++11, C++ has its own memory model and synchronization operations. The C++ memory model, much like the other memory models we saw in the videos, defines the ordering of memory operation and which orderings are respected under which conditions. These are independent of the underlying hardware, with the compiler being responsible for mapping the language model to the underlying hardware model. This is done on purpose, as multi-threaded programs that are written correctly in modern C++ should be able to be executed correctly on a variety of hardware, regardless of the memory model of the hardware. In practice, this means that the compiler has to emit appropriate synchronization operations (e.g., fences or atomics) depending on the targeted hardware. For example, if you recall, hardware using the TSO memory model respect all orderings except for the "store -> load" ordering, allowing younger loads to bypass older stores. If the C++ memory model specifies that a particular "store -> load" pair needs to be ordered in program order, then the compiler has to emit a fence. On the other hand, the compiler does not need to emit any fences for any "load -> load" pairs that need to be ordered in program order, as these are already respected by default on TSO. On a different processor, with a different memory model, such as RC, then the compiler might need to emit even more fences to ensure that the program is executed according to the C++ memory model.

C++ Synchronization

So what is "synchronization" in the context of the C++ memory model? Before we can answer that, we need to consider what it means for two operations to be **ordered**. Let's take the following example:

```
int a = 0;
bool updated_a = false;
//Thread 1                    // Thread 2
a = 1;                        while (!updated_a) {
updated_a = true;                 // wait
                              }
                              print(a); // What will this print?
```

What will Thread 2 print in this case? You should recall from Module 2.1 that without specifying a memory model, it is not possible to know what the answer will be. This is because, without specifying a memory model, there is no way of knowing what the **order** of the memory operations will be. For example, on an RC model, it is possible for the store to `updated_a` to happen before the store to `a` (e.g., imagine that the store to `updated_a` hits in the cache, while the store to `a` misses in the cache), in which case Thread 2 will print "0". Under RC it is also possible (i.e., it can randomly happen) that all stores and loads just happen in program order, and Thread 2 will then print "1". On the other hand, under TSO, we know that stores are always **rdered** with respect to other stores (i.e., we know that the store to `a` in Thread 1 will **happen efore** the store to `updated_a`) and we know that loads are always **ordered** with respect to other loads (i.e., we know that the load to `updated_a` in Thread 2 will **happen before** the load to `a`). Thus, we know that Thread 2 will always print "1".

Coming back to C++, the memory model in the language is very similar to RC (with some small differences which we will not go into here) with an "SC-for-DRF" guarantee. "SC-for-DRF" stands for "Sequential Consistency for Data Race Free" and means that as long as there are no data races then the code will behave as if we have an SC memory model. Data races themselves happen when we have multiple threads accessing the same data, at least on of them is modifying the data, and there is **no order** established between the threads. Since the underlying memory model is RC, there is, by default, no order established between any loads or stores, so any loads or stores accessing the same data without first explicitly establishing an order constitute a data race. For example, in the code above, there are data races between Thread 1 and 2 when Thread 1 updates `a` and Thread 2 reads `a` without first establishing some sort of order, as well as when Thread 1 updates `updated_a` and Thread 2 reads `updated_a` without first establishing some sort of order. And how do we apply an order across two threads? By using the synchronization operations found in the language, such as locks, atomics, and fences:

```
int a = 0
atomic_bool updated_a = false;
//Thread 1                          // Thread 2
a = 1;                              while (!updated_a.load()) {
updated_a.store(true);                  // wait
                                    }
                                    print(a); // Will always print "1"
```

The atomic store and load operation will synchronize the two threads and establish the order between the operations, which will guarantee us that the result will always be "1".

It is important to note here, that writing code that is not properly synchronized is considered a bug in C++ and the resulting behavior is undefined, so it should never be done. Also, using volatile variables DOES NOT constitute synchronization modern C++ (or in modern C). You can read more about the C++ memory model here: **https://en.cppreference.com/w/cpp/language/memory_model,** (https://en.cppreference.com/w/cpp/language/memory_model) but it all boils down to this simple guideline: **If you have data that is shared among threads, and at least one thread is going to modify that data, either make the shared data atomic or use locks.** Of course, it is possible to further optimize the code without introducing any bugs, this is just a simple guideline to make writing multi-threaded applications easier.

C++ Locks and Atomics

As already mentioned, there are three main synchronization methods in C++: locks, atomics, and fences. Fences are generally not recommended, as they can be hard to use correctly, so we will not discuss them for the basic part of the lab. Locks are much like the locks discussed in the videos, with the main lock type being `std::mutex,` here: **https://en.cppreference.com/w/cpp/thread/mutex** (https://en.cppreference.com/w/cpp/thread/mutex) . Locks provide "mutual exclusion", where only one thread at a time is allowed to access the code between the lock and the unlock operations (referred to as the "critical section"). Atomics on the other hand do not guarantee mutual exclusion, they only guarantee that the operation will happen atomically, i.e., it will happen as if it was one single operation uninterrupted by other threads.

To use atomics in C++, we have to use the appropriate **atomic types (https://en.cppreference.com/w/cpp/atomic/atomic)** . This can be done either by wrapping an existing type in an `std::atomic` type (as a template parameter) or by using one of the predefined atomic types, such as the `atomic_bool` type in the example in the previous subsection. You can find a concise list of all the standard defined types, as well as the functions around them, here: **https://en.cppreference.com/w/cpp/atomic/atomic. (https://en.cppreference.com/w/cpp/atomic/atomic)** **An important thing to notice here is that a variable can either be atomic or non-atomic and you cannot mix between atomic and non-atomic operations on the same variable.**

You will also notice that most atomic operations come with two variants, a "normal" and an "explicit" one. The difference between the two is that in the explicit variant it is possible to relax the memory order established by the atomic operation. The explicit variant is reserved only for expert programmers that fully understand the programming model and the synchronization patters in their code, so they should generally be avoided. For the purposes of this lab, you should only use the normal variants.

You might also notice that the atomic types have overloaded operators that allow them to be modified without explicitly calling any functions. For example, instead of using the `fetch_add` function, it is possible to simply use the `++` operator. **If you use these overloads, please add comments in your code at each place where an atomic operation is happening** (hint: It's everywhere the atomic variable is being used.) with what sort of operation it is.

Prerequisites

You will need a reasonably modern C++ compiler, with support for C++11, atomics, threads, and mutexes. Also, you will need a machine with at least two cores to run the code on. We expect the code to be portable across Linux, Windows, and MacOS, with either gcc or clang, but we have not tested all of them, so please let us know if you encounter any compilation issues. If you have issues running code on your machine, or if you do not have a machine that meets the requirements, feel free to use any of the **machines that the department provides (https://www.it.uu.se/datordrift/maskinpark/linux)** . These can be accessed remotely via ssh.

Note: On Windows machines you might have to modify the Makefile to add `.exe` in the end of the binaries in the `test` and `clean` targets, if your compiler automatically generates files with the .exe extension. You might also need to replace `rm -rf` with `del /q` .

The files for the lab are available on Studium under "**Assignments 2 (https://uppsala.instructure.com/courses/94980/files/7996066?wrap=1) (https://uppsala.instructure.com/courses/94980/files/7996066/download?download_frd=1) → Lab 2 Files (https://uppsala.instructure.com/courses/94980/files/7996066?wrap=1) (https://uppsala.instructure.com/courses/94980/files/7996066/download?download_frd=1)** ".

Part A: Using Synchronization Primitives

In this part of the lab, we will be using the existing synchronization primitives available in modern C++ to implement an *atomic counter.* An atomic counter is a type that holds a value which can be incremented or decremented safely by several threads at the same time. For this lab, the underlying type of the counter is a simple integer. In the lab files you will find the following files:

- **atomic_counters.hpp:** This is the main header file that contains all the different implementations of the atomic counter. There is a base `atomic_counter` abstract class, which defines the interface of the atomic counter, and a number of concrete specializations that implement the atomic counter using different synchronization primitives. The header file only contains the class definitions and not any member definitions (i.e., the function implementations) of any of the classes. These can be found in the `.cpp` files listed below.
- **atomic_counter_nosync.cpp:** This file contains the definitions for the `atomic_counter_nosync` class, which implements an atomic counter without using any synchronization primitives. You should take a look at this file, but you should not modify any of the member functions in it.
- **atomic_counter_lock.cpp:** This file contains the member function definitions for the atomic counter that should be implemented using locks. You will find that most of the functions in this file are marked with `TODO` notes, indicating that you should modify them. Take a look at the header file (`atomic_counters.hpp`) to see what the class definition looks like and what members it contains.
- **atomic_counter_atomic_incdec.cpp:** In this file you will find the member function definitions for the atomic counter using atomic increment and decrement operations. Once again, here you will need to modify the existing code using the appropriate synchronization primitives.
- **atomic_counter_atomic_cas.cpp:** This is similar to the previous file, only now we want to use atomic compare and exchange instead of atomic increment and decrement operations.
- **test_atomic_counter.cpp:** This file contains the `main` function and performs tests on all the different atomic counter implementations. Feel free to take a look here if you want to see how things work, but **do not modify anything in this file**.

You will find `TODO` markers in all the places that you need to modify in order to complete the various atomic counter implementations. **Do not change any of the code that is not marked with `TODO`** without consulting with one of the TAs first.

For compiling and testing your code, a Makefile is provided for you. Simply invoking the `make` command in the source directory will compile all the source files and run the tests. If you only want to compile the files, without running the tests, then invoke make with the "all" target stead: `make all`. The program that performs the tests is called `test_atomic_counter` and can be nvoked through the command line with an additional argument that specifies which implementation to test. **Do not modify the Makefile without consulting with one of the TAs first.**

Tasks

1. Compile and run the atomic counter test using the nosync counter implementation: `./test_atomic_counter nosync`. **Do not modify the nosync implementation.** Does it work? Why or why not?
2. Implement the **lock-based** atomic counter in `atomic_counter_lock.cpp`. You will find `TODO` marks in the functions that you need to change. You should also read the comments in the base class `atomic_counter` in the `atomic_counters.hpp` file and also take a look at the `atomic_counter_lock` class definition as well. Compile (using the makefile) and run the test: `./test_atomic_counter lock`. Does it work? Why or why not?
3. Implement the **atomic inc/dec-based** atomic counter in `atomic_counter_atomic_incdec.cpp`. You will find `TODO` marks in the functions that you need to change. Use the **`fetch_add`** and **`fetch_sub`** functions available in C++. For this, you will also need to modify the header file, to make sure that the `m_value` variable if of the correct type. Compile (using the makefile) and run the test: `./test_atomic_counter atomic_incdec`. Does it work? Why or why not?
4. Implement the **atomic CAS-based** atomic counter in `atomic_counter_atomic_cas.cpp`. Use the **`compare_exchange_strong/weak`** functions in C++. Compile (using the makefile) and run the test: `./test_atomic_counter atomic_cas`. Does it work? Why or why not? How do the `compare_exchange` functions work?
5. Observe the performance (in operations per second) of the different tests and rank them from fastest to slowest. Why do they end up in this order?

For the atomic inc/dec and CAS based implementations, remember to either use explicit method calls for atomic operations or add comments in any places the atomic operator overloads provided by C++ are being used.

Part B: Implementing Locks

In Part A we used locks and atomics to synchronize the accesses to the `m_value` variable in the atomic counter implementation. In this part, we will instead use atomics to implement our own locks. The relevant files for this part are the following:

- **user_locks.hpp:** This is the main header file containing all the class definitions. Much like in Part A, you will need to modify some of them here, but the code for implementing the lock and unlock operations resides in the `.cpp` files.
- **user_lock_mutex.cpp:** Here you can find an implementation of the lock using a mutex from the standard library. You do not need to implement anything in this file, this implementation is provided only as a reference.
- **user_lock_dekker.cpp:** Here you will have to implement a lock using Dekker's algorithm.

Dekker's Algorithm

Dekker's algorithm can be used to implement mutual exclusion between two threads. In the videos we saw a simple version that is designed to allow only one of the two threads to enter the critical section. Here you can find a version which can be used to implement an actual lock that works with two threads, i and j:

```
// Lock
flag[i] ← true
while flag[j] == true do
    if turn != i then
        flag[i] ← false
        while turn != i do
            // do nothing
        end while
        flag[i] ← true
    end if
end while

// Critical section goes here

// Unlock
turn ← j
flag[i] ← false
```

The `turn` variable should be initialized to 0 and the `flags` should be initialized to false.

Tasks

1. Compile and run the `std::mutex` based lock: `./test_user_lock mutex`. Does it work? Why or why not.
2. Implement mutual exclusion using Dekker's algorithm in `user_lock_dekker.cpp.` Do not use any locks, instead use atomic operations where necessary. Test your implementation: `./test_user_lock dekker`. Does it work? Why or why not.
3. What is the memory model of the language? What about the memory model of the underlying hardware? Which memory model are we programming for here?
4. What if we had a different programming model?
5. Observe the performance of the two different lock implementations. Which one is faster? Why?

< Föregående (https://uppsala.instructure.com/courses/94980/modules/items/1296779)

Nästa > (https://uppsala.instructure.com/courses/94980/modules/items/1296781)