TDDD56 Multicore and GPU computing
Lab 2: Non-blocking data structures

August Ernstsson, Nicolas Melot
`august.ernstsson@liu.se`

November 18, 2020

1 Introduction

The protection of shared data structures against inconsistent states, caused by several threads concurrently using the same structure is usually achieved through locks or semaphores. Semaphores and locks synchronize threads in order to prevent them from destroying a shared data structure by modifying it concurrently. Using locks, only one thread can perform an operation at a time and another thread has to wait for this operation to be finished before it can perform its own operation. This technique guarantees the data structures to be constantly consistent. However if many threads use heavily the same structure at the same time, there may be long queues of threads waiting for getting access to the same resource, making these threads spending more time waiting than performing actual computation.

Non-blocking data structures help at reducing this effect, by eliminating as much as possible exclusive accesses to resources. Instead of locks, they rely on atomic instructions that check the operation to perform before committing them. If they can be performed safely, operations are done and a success is reported, otherwise the instruction reports a failure. Data structures protected using non-blocking synchronization may be vulnerable to the ABA problem, which essentially happen when the structure changed and was rolled back to its initial state between to access from the same thread. This may or may not be a problem depending on applications. If the underlying hardware architecture doesn't provide suitable instructions, one can use locks to implement the same atomic operations. However, this might not provide the same speedup, and they are still vulnerable to the ABA problem.

This laboratory focuses on the case of a shared stack and how locks and non-blocking synchronization affect its performance. Although shared use cases of stack may not be obvious, they can be employed to implement task stealing, in a case where last job to enter the queue is the most prioritary to run. We use the hardware instruction CAS (Compare-And-Swap) to achieve non-blocking synchronization of the task and measure the performance improvement. We illustrate the ABA problem through an optimized task that does not destroy its element after popping them, but store them in order to reuse them next time another job need to be enqueued. Such strategy can generate speedup by saving costly calls to $malloc()$ and $free()$. The work consists in implementing a classic stack and protect it using pthread locks. In a second step, we modify the stack in order to use non-blocking synchronization instead. The third step consists in using pthread locks to force the parallel program to execute in such a way that triggers the ABA problem, and observe evidence that ABA actually happened.

2 Getting started

2.1 Installation

Fetch the lab 2 skeleton source files from the CPU lab page and extract them to your personal storage space.

2.2 Source file skeleton

This section describes how to use and compile the skeleton source files provided and introduce a few key points you need to know.

2.2.1 Compiling and running

The skeleton can be compiled in three different modes you can enable or disable by passing variables when calling make:

Measuring: This makes the executable to compute one picture and quit without saving it. It also makes the program to check time (in seconds and nanoseconds) before and after computating, and display these values on the terminal after completion and before exiting. This mode is used by batch and plotting scripts to generate many values, analyse and plot them. You can also use these numbers if you want to interpret them yourself. Call make with the variable $MEASURE = n$ where $n = 1$ to measure the performance of *pop* operations and $n = 2$ for push operations (example: *make MEASURE=1*).

Testing: This mode runs unit tests you wrote yourself and makes sure your stack work correctly sequentially, in parallel using pthread lock and in parallel using non-blocking synchronization. Using this mode and implementing tests is optional, although strongly recommended. Call make with the variable $MEASURE = 0$ (*make MEASURE=0*).

Make can also take other useful options through other variables passing:

NB_THREADS: Call make with $NB\_THREADS = n$ where n is a non-negative integer. It instructs make to compile a multithreaded version running n threads. $n < 2$ makes little sense as we want to test the behavior of the stack when at least 2 threads use it concurrently.

NON_BLOCKING: Call make with $NON\_BLOCKING = n$ where $n \in [0,1,2]$. It selects and compile one synchronization method among lock-based (0), software-based CAS (1) and hardware-based CAS. Note that you *must* use the *NON_BLOCKING* symbols and preprocessor directives (*#if #then #else #endif*) in your code in order to let the compiler to select and compile the portion of code corresponding to the synchronization method it aims at using.

MAX_PUSH_POP: Call make with $MAX\_PUSH\_POP = n$ where n is a non-negative integer ($n = 5000$ by default). This affects the maximum amount of *PUSH* or *POP* operations in a performance test. The higher the value, the longer the tests, the more reliable and obvious the results.

2.2.2 Structure

The skeleton provides function prototypes and an assembly implementation of hardware CAS. You need to fill in the gap for performance tests and actual stack implementation. You can also optionally implement unit tests to make sure your stack works as expected.

test.c The program starts here. Depending on the mode you compiled the source with, it may run unit tests or performance tests. You need to implement here your performance test. You may (encouraged) also implement unit tests. Below are described a few helper functions and a hint on how to use them; they are *not* used with the performance tests:

1. $test\_init()$ Runs exactly once before the batch of tests starts to run. Ideal place to initialize the global test environment.
2. $test\_setup()$ Runs before each unit test is started. It is the preferred place to initialize a stack in a well-know safe state.
3. $test\_teardown()$ Runs after each unit test finishes. You can destroy the stack you tested here
4. $test\_finalize()$ Runs after all tests have been all completed. Clean your testing environment here.

The skeleton further provide unit test stub functions such as *test_push_safe()* and *stack_pop_safe()*. See in *main* function on how the skeleton runs them through *run_test()*. Each unit test function is run by one thread only and returns 0 if the unit test is considered successful and non-zero otherwise. You are encouraged to provide an implementation to these function and create others so you can test if your stack behaves as expected, although this is not required. For example, can your stack push or pop properly a single element? Are the thread-safe push or pop variants also correct when using one thread? Are they still when using more than one thread? You can take inspiration from *test_cas()* on how to spawn the threads required for a unit test you design. Note that if the skeleton is compiled with the symbol *MEASURE* set to non-zero, then the skeleton does not run the unit tests but it runs the performance test (also defined in *main()*) instead.

stack.c Actual implementation of your stack. Provide an implementation to allocate, initialize your stack and perform *push* and *pop* operations on it. You need later to use the *NON_BLOCKING* preprocessing symbols to separate lock-based, software cas-based and hardware cas-based variant of these operations. You are also encouraged to fill in the function *stack_check* to assert ($asserts()$) properties of the stack to make sure anytime (in your unit tests) that your stack is valid.

non_blocking.c Provides an assembly implementation of hardware cas. You can also implement a software-based cas into to dig further experimentations around synchronization variants (optional).

3 Before the lab session

Before coming to the lab, we recommend you do the following preparatory work:

- Write an explanation on how CAS can be used to implement protection for concurrent use of data structures
- Sketch a scenario featuring several threads raising the ABA problem
- Implement all the algorithms required in sections 5 and 6 ahead of the lab session scheduled, so you can measure their performance during lab sessions. Use the preprocessor symbol NON_BLOCKING to take the decision to use either a lock-based stack, a software CAS-based stack or a hardware-CAS-based stack (respectively values 0, 1 and 2). This value is known at compile time and can be handled using preprocessor instructions such as *#if-#then-#else*; see skeleton for example.
  The stack is required to be **unbounded** in capacity (within the constraints of the system), i.e., you *cannot* implement the stack as an array-based data structure.

4 Hints

- Make sure your performance test has no chance to trigger the ABA problem; the results may not be meaningful if the stack tested is corrupt.
- Remember that *malloc*() and *free*() operation manage a global memory space and use a global, thread-shared queue to manage it, managed by locks. This can affect the performance you measure. **Your performance tests should not trigger malloc or free for every push or pop operation.** You will need to find a way to manage your own memory without synchronizing access.
- Your stack implementation has to be unbounded, so design it linked-list style (no ring buffers or similar).
  However, you are allowed to initialize the stack with a finite number of pre-allocated list elements for the purpose of testing in this lab. (How would you solve this in a practical scenario?)

5 During the lab session

Take profit of the exclusive access you have to the computer you use in the lab session to perform the following tasks

- Measure the performance of a lock-based concurrent stack and generate a plot showing the time to perform concurrently a fixed amount of push, and separately a fixed amount of pop, as a function of number of threads involved.
- Measure the performance of a CAS-based concurrent stack and build a graph featuring the same scenario as above. Make sure the performance tests avoids the ABA problem (push or pop only, no garbage collection, etc).

  Hint: it may happen that you obtain curves that do not fit your expectations. Think about what the threads you implemented mostly do: do they really push and pop in parallel? What about overhead? If these thoughts are not sufficient to explain the curves you see, try to have threads to run the exact same test using one different stack each; no synchronization should be needed and the curve should show a decreasing time with number of threads. Once you have the curves you expect, run the test with a shared stack again and use the clues above to form an hypothesis you can propose for the demo.

6 Lab demo

Demonstrate to your lab assistant the following elements:

1. Explain how CAS can implement a safe data structure sharing between several threads
2. Explain the scenario raising the ABA problem you have implemented
3. Execute a multi-threaded implementation of the *test_aba*() unit test program where several threads use CAS to push and/or pop concurrently to the same shared stack. Synchronize explicitly this program using pthread locks or semaphores in order to make the threads to synchronize each other until the ABA problem arises. You can clone and specialize your *push*() or *pop*() implementations for each thread, with semaphores at the right place to simulate unlucky thread switches leading to an ABA occurrence. Detect the ABA occurrence and return 1 of ABA is detected, 0 otherwise. Show *printf(...)* traces along the synchronization routines you use in order to explain the important steps leading to the situation where the ABA problem rises.

4. Show compare and comment the performance of both lock-based and hardware CAS-based concurrent stacks, as a function of number of threads.