# EDA 284

# Lab 2 – Q1 2018

The purpose of this lab exercise is to acquaint you with synchronization primitives.

For this lab we are going to use **perf** to get some insight into the runtime characteristics of each version of your programs.

Go through the steps in this lab one by one and answer all questions in your lab report.

**Part A.**

**1.** Download the code from ping-pong and look at lock1.cpp, this will be the code you will build on for this lab. This is a simple multi-threaded program where each thread increments a shared variable a given number of times. Each thread must acquire a lock before updating the shared variable, also all threads must reach a barrier before they start updating the shared variable to make sure all threads start at approximately the same time.

In lock1.cpp the lock is implemented as a class which is just a wrapper around a C++ mutex like the one we used in the first lab. Examine the code of this example, make sure you understand how the lock works and run it for a variable number of threads and final values for the shared variable.

To compile use: **g++ -std=c++11 -pthread lock1.cpp -o lock1**

to run use: **./lock1 NUM_THREADS "final value"**

for example: **./lock1 2 40000000**
runs the program for 2 threads and the final value of the shared variable is going to be 40.000.000. The program automatically divides the work per thread so that the total work done by all threads is the same regardless of the number of threads. For example if you run **./example 2 40000000** each thread will increment the shared variable 40000000/2 = 20000000 times. If you run
**./lock1 4 40000000** then each thread will increment the variable 40000000/4 = 10000000 times.

Run the code for 1,2, and 4 threads using **perf** to measure the runtime, number of instructions, execution cycles, L1 data cache accesses, and L1 data cache misses (refer to lab2 PM for perf examples). Note down the results for your lab report.

**2.** For your next task you are going to use atomic primitives provided by the processor to implement a lock by yourselves. The atomic primitive we are going to use is Compare and Swap (CAS). Compare and Swap compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value. To use CAS we are going to use a gnu atomic built-in:

**bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval)**

example:
**int lock;**
**bool result = __sync_bool_compare_and_swap (&lock, 0, 1)**

This call will return true if the lock was 0 and it was successfully updated to 1, false otherwise. To implement a lock you can use this built-in in a while loop and call it until it returns true. Conventionally if lock is “0” it means it is unlocked, if it is “1” it means it is locked.

Change the eda283_lock class to implement a lock with this builtin and without using the mutex that is already there. Do you need CAS for both locking and unlocking? Why? Why not?

Implement your own lock as suggested above and save the file as **lock2.cpp,** run the code with the new lock implementation for the same configurations as (1) and use **perf** in the same way to measure the same statistics, note these statistics down for your report. Is the new lock implementation better of worse that the **mutex** in **lock1.cpp**? Why? Why not? And for which configurations (number of threads)?

**3.** In (2) you implemented a spin-lock, that is the threads insist on trying to get the lock all the time in your while loop. To make this implementation more efficient we are going to make the threads yield the processor if **CAS** does not succeed. For that we are going to use

**this_thread::yield();**

In the eda283_lock class, in the CAS while loop. Notice that although this call is located in the lock class the yield() function is going to apply to the calling thread.

Save this implementation as **lock3.cpp**. Run the same experiments and use perf as before to gather statistics and see what happens in the runtime of the application when using yield(). Is it better or worse, and for what number of threads? Explain your results in your report.

**4.** For this task you will use the **test and test and set** technique as described in class. For locking you will use two nested loops, in the first loop you will use CAS to try and set the lock to 1 and if that fails the body of the loop will be executed where you will use another while loop that checks the lock without using CAS or any other atomic operation and only if the value of the lock is zero it will exit and thus the CAS in the first loop will try again. This technique saves some “expensive” atomic operations, can you verify that from the statistics you get from **perf**? Measure the runtime characteristics of this implementation as you did for the previous and add the results to your report.

**5.** Gather your statistics for all previous implementations of locks and plot the statistics you gathered for each implementation for 1, 2, and 4 threads. Produce different graphs for runtime, number of instructions, L1 data cache accesses , and L1 dcache misses. The Y axis of your charts should be the statistics and the X axis should be the number of threads, use four different columns, one for each lock implementation.

**Part B (Optional).**

For this part you are going to implement the barrier on your own. Download barrier1.cpp and look at the code. The eda283_barrier class is just a wrapper around a pthreads barrier which implements the following two functions

**init(int num_threads)** which sets the number of threads the barrier should wait for
and
**wait()** which is called by each thread that has reached the barrier and only returns when all threads have reached it

**HINTS:**

To implement the barrier you are going to need to store in the class the number of threads the barrier should wait for which will be set by the **init** function and an additional variable which will store the number of threads that has reached the barrier every time. This variable must be incremented every time a thread calls **wait()** and **wait()** will return only once all threads have reached the barrier. You are going to have to use a lock to “protect” that variable and any other variable which is modified by more than one thread.

Are the variables mentioned above enough for the barrier implementations? What happens when the barrier is reached by all threads and wait() returns and which thread must reset the barrier to wait for the same number of threads every time? What happens if the barrier is reset by one thread before all threads have returned from the wait() function?

You are free to look up the internet for solutions but try to think of the problems and solutions to these problems by yourself first.