CS351 Lab #EC2

Benchmarking Synchronization Mechanisms

Instructions:

- *Assigned date: Thursday November 19th, 2020*
- *Due date: 11:59PM on Sunday December 13th, 2020 [firm deadline, no extensions; no submissions after this deadline will be accepted]*
- *Extra Credit: 30 points*
- *This lab must be done individually*
- *Please post your questions to the Piazza forum*
- *Only a softcopy submission is required; it will automatically be collected through GIT at the deadline; email confirmation will be sent to your HAWK email address; submissions will be graded based on the collected submissions at the deadline, unless an email to the TAs at cs351-ta-group@iit.edu with the subject "[CS351] homework submission is delayed" is received; when a student is ready to have their late assignment graded, they must send another email to the TAs at cs351-ta-group@iit.edu with the subject "[CS351] late homework submission is ready"; late submission will be penalized at 5% per day*

1 Your Assignment

In order to program for shared memory systems using multi-threading, threads need to be synchronized. Various thread synchronization mechanisms exist which ensure that threads do not simultaneously execute a critical section of the program. Many languages provide high level abstractions for synchronization to ease parallel programming. Common synchronization mechanisms include mutexes (mutual exclusion locks), semaphores, reader/writer locks and condition variables. Mutex is a mutual exclusion lock which ensures exclusive access to the shared resource. Spinlock is a type of lock which waits in a busy loop if lock cannot be acquired. Atomics operations are instructions supported by hardware and they lock the memory bus to access the shared resource. These operations are inherently atomic and have limited support for data types on various architectures. Semaphores is a type of mutual exclusion where a thread can wait to get access to the critical section or do a post so other threads can get access.

The primary focus here is to analyze the cost of low-level thread synchronization mechanisms and for this purpose, you will benchmark *pthread_mutex_lock*, *pthread_mutex_unlock*, *sem_wait/sem_post*, *fetch-and-add*, and *spin_lock/spin_unlock* to measure latency. *spin_lock* and *spin_unlock* should be implemented using *compare_and_swap* atomic primitive. *Fetch-and-add* is supported by x86 architectures using '*lock xadd*' instruction. These benchmarks should be obtained by running a tight loop of 1 billion operations and collecting the aggregate of the results. Each iteration acquires the lock, increments a shared integer and releases the lock, excluding fetch-and-add which performs an increment operation atomically. You will perform a weak scaling study, unless otherwise noted; this means you will set the amount of work per thread (e.g. the number of operations to evaluate in your benchmark), and keep the amount of work per thread fixed as you increase the number of threads.

This project aims to teach you how to benchmark four different synchronization mechanism. You must use the C programming language and can use abstractions such as PThreads. Libraries such as STL or Boost cannot be used. You can use any Linux system for your development, but you must make sure it compiles and runs correctly on fourier. The performance evaluation should be done on your own computer in a

Linux environment (e.g. think back at Lab #0). Given the large class size, we encourage all students to avoid using fourier for anything but testing functionality on small datasets.

Your syncbench program should support the following benchmarks:

1. Mode:
    a. vanilla: increment an integer counter with no synchronization mechanisms
    b. mutex: use a mutex to protect an integer while it is being incremented from 1 to N concurrent threads
    c. semaphore: use a semaphore to protect an integer while it is being incremented from 1 to N concurrent threads
    d. spinlock: use a spinlock to protect an integer while it is being incremented from 1 to N concurrent threads
    e. atomic: use an atomic to allow the thread safe incrementing of an integer from 1 to N concurrent threads
2. Size:
    a. Small: this should be set to 100 million operations (10^8)
    b. Large: this should be set to 1 billion operations (10^9)
3. Threads:
    a. 1: Use 1 thread to execute your benchmark on the specified number of operations
    b. 2: Use 2 threads to execute your benchmark in parallel on the specified number of operations per thread
    c. 4: Use 4 threads to execute your benchmark in parallel on the specified number of operations per thread
4. Metrics:
    a. Throughput: measure the rate at which the counter can be incremented per second using the large size (1 billion operations); throughput should be calculated for each thread individually and all the results should be aggregated to get the final throughput value for the experiment.
    b. Latency: measure each individual increment operation and store it in memory using the small size (100 million operations); see discussion for fine-grained measurements for more information on how to measure latency

Fine grained measurements

You will not be able to use the timing mechanisms for the latency part of this evaluation due to the short duration of the synchronization mechanisms. On x86 architectures, latency is measured in CPU cycles using RDTSCP instruction for start time and RDTSC + CPUID instruction for the end time. RDTSCP is a serializing instruction and it prevents instruction reordering around the call. CPUID is also a serializing call and when it follows RDTSC instruction, it prevents any future instructions to be executed before timing information is read. The combination of these two timing functions gives the most accurate results for latency. Timing on ARM could be quite different than x86, so be careful if you are trying to run on anything other than x86. Time base register counts cycles at a fixed lower frequency and needs to be calibrated to convert the value to actual cycles at CPU clock frequency.

Fill in the table 1 below for throughput using the large size; report the average throughput in ops/sec:

Mode	Threads	Measured Throughput
Vanilla	1	
Mutex	1	
Semaphore	1	
Spinlock	1	
Atomic	1	
Vanilla	2	
Mutex	2	
Semaphore	2	
Spinlock	2	
Atomic	2	
Vanilla	4	
Mutex	4	
Semaphore	4	
Spinlock	4	
Atomic	4	

Fill in the table 2 below for latency using the small size; report the average latency in cycles and nanoseconds:

Mode	Threads	Measured latency (cycles)	Measured latency (nanoseconds)
Vanilla	1		
Mutex	1		
Semaphore	1		
Spinlock	1		
Atomic	1		
Vanilla	2		
Mutex	2		
Semaphore	2		
Spinlock	2		
Atomic	2		
Vanilla	4		
Mutex	4		
Semaphore	4		
Spinlock	4		
Atomic	4		

Other requirements:

- You must write all benchmarks from scratch. Do not use code you find online, as you will get 0 credit for this assignment.
- All of the benchmarks will have to evaluate concurrency performance; concurrency can be achieved using threads and processes. For this assignment, you must use multi-threading. Use weak scaling in all experiments, unless it is not possible, in which case you need to explain why a strong scaling experiment was not done. Be aware of the thread synchronizing issues to avoid inconsistency or deadlock in your system.

- All your benchmarks can be run on a single machine. You should run these benchmarks on your own system under Linux. Make sure your work compiles and can run the small workloads on fourier.
- Not all timing functions have the same accuracy; you must find one that has cycle accuracy.
- Since there are many experiments to run, find ways (e.g. scripts) to automate the performance evaluation.
- There are likely some compiler flags that will help you execute your benchmarks faster.
- You might find this technical report useful: http://datasys.cs.iit.edu/publications/2017_IIT-oral-xtask.pdf
- No GUIs are required. Simple command line interfaces are expected.

2 What you will submit

When you have finished implementing the complete assignment as described above, you should submit your solution to your private git repository. Each program must work correctly and be detailed in-line documented. You should hand in:

1. **Source code and compilation (70%):** All of the source code in C and Bash; in order to get full credit for the source code, your code must have in-line documents, must compile (with a Makefile), and must be able to run a variety of benchmarks through command line arguments. Must have working code that compiles and runs on fourier.
2. **Report / Performance (30%):** A separate (typed) design document (named labEC2-report.pdf) describing the results in a table format. You must evaluate the performance of the various parameters outlined and fill in the 2 tables specified to showcase the results. You must summarize your findings and explain why you achieve the performance you achieve, and how the results compare between the various approaches.

To submit your work, simply commit all your changes to the Makefile, syncbench.c, runbench.sh, and push your work to Github. You can find a git cheat sheet here: https://www.git-tower.com/blog/git-cheat-sheet/. Your solution will be collected automatically at the deadline. If you want to submit your homework later, you will have to push your final version to your GIT repository and you will have let the TA know of it through email. There is no need to submit anything on BB for this assignment. If you cannot access your repository contact the TAs.

No late submissions will be accepted.