Names: Craig Li, Prerak Patel
NetIDs: craigli, pjp179
ilab Server used: kill.cs.rutgers.edu
CS416 - Operating Systems

Project 3 Report

Introduction

Project 3 allowed us to explore the foundations of concurrency by implementing four different approaches toward a counter method. Through the creation of each of these different approaches, we were able to explore the indeterminate results of unsafe thread practices, as well as the determinate results that were a result of following thread safe practices such as mutual exclusion and synchronization. By vastly narrowing the scope of Project 3 to only focus on the effects of concurrency, as opposed to the much more involved implementation of Project 2's thread library, we are able to focus much further on the fundamental issues with concurrency, and their implementation in a multi-core system. To illustrate our understanding, we have included our data from each implementation of the counter, as well as provided answers to the questions below.

1. **Why does Naive Counter have a huge difference from the correct value?**

   The Naive Counter exhibits a large difference from the correct value due to the underlying nature of concurrency, as well as the lack of thread-safe practices used in the program. In the Naive Counter, we are simply incrementing the counter by one over 16,777,216 (1UL<<24) times in each thread. However, our problem exists in the scheduling of each thread. Without necessary thread-safe code such as mutexes or barriers, the scheduler can interfere with our operations, leading to indeterminate behavior, which is displayed as the huge difference from the correct value. The incrementation of the counter variable is represented in machine code as multiple assembly instructions, which may look like this:

```
0x0000555555555134 <+11>:    mov    -0x14(%rbp),%eax
0x0000555555555137 <+14>:    add    $0x1,%eax
0x000055555555513a <+17>:    mov    %eax,-0x4(%rbp)
```

   The value of counter is loaded into register eax, adds 1 to the value of eax, and then eax is moved back into memory at the same address. Our problem occurs when this set of three commands is interrupted by the OS scheduler, which can interrupt at any point due to the non-atomic nature of the addition operation. Let us assume a hypothetical situation where the counter's value is 10. Should the thread be interrupted while in the middle of these instructions (let's say after the first instruction, but before the second), its state is saved in its TCB, and control is switched to a different thread, which then enters the same piece of code (all threads are incrementing the counter variable). Since the first thread has not completed its addition operation, the value of counter is still 10, and the second thread reads this value, adds 1, then writes 11 back into the value of counter. When thread 1 resumes, it still holds the value of counter as 10 in its registers, and completing the addition operation will result in thread 1 writing the value of 11 into the counter. Now we have a situation where two increments have occurred, yet the value of the counter has only increased by one. This is called a race condition, and is one of the main problems with concurrency, but one that can be easily remedied by implementation of mutual exclusion.

   This race condition is exacerbated in our code due to the large number of increments that we have for the counter variable. Each thread is to increment the counter 16,777,216 times, and due to the speed at which each thread is swapped out, the aforementioned race condition occurs an incredible amount of times, leaving our actual counter value far from our predicted value.

Names: Craig Li, Prerak Patel
NetIDs: craigli, pjp179
ilab Server used: kill.cs.rutgers.edu
CS416 - Operating Systems

Project 3 Report

2. **Why would Atomic Counter be superior to Naive Counter Plus?**

   Our implementation of Atomic Counter would be superior to our implementation of Naive Counter plus due to the differing approaches to mutual exclusion demonstrated in each of the files. In Naive Counter Plus, we use pthread_mutex_lock() and pthread_mutex_unlock() to accomplish the goal of mutual exclusion, guarding the addition operation so that it cannot be interrupted. However, in Atomic Counter, we can use atomic operations to accomplish our mutual exclusion and the addition operation without the use of locks. Atomic operations create a full memory fence to ensure that neither the compiler nor the executing processor will be able to re-order memory operations across the atomic operation, which also ensures that we do not have any race conditions. In this respect, atomic operations are by default put into a memory_order of memory_order_seq_cst, which not only preserves the order that the threads access the critical section, but allows each thread to see the effects of the previous atomic operation. Unlike the pthread locks used in Naive Counter Plus, this does not actually block threads from executing, but merely establishes an order while the threads in Atomic Counter can keep running, making our Atomic Counter run much faster than Naive Counter Plus.

3. **Why is Atomic Counter still not able to achieve better performance than Naive Counter?**

   Our Atomic Counter is still not able to achieve better performance than our Naive Counter due to the innate protections that are included with using an atomic operation. Though we are “reducing” the number of machine instructions from 3 to 1, we are still providing protections so that the memory accesses from each thread are maintained in a specific order, which does not happen with Naive Counter. Due to this, Naive Counter interleaves its own instructions at the maximum possible speed, while Atomic Counter has to execute in a specific order, leading to a much faster runtime for Naive Counter than Atomic Counter. HHowever, this comes at the cost of correctness, as we know that our output for Naive Counter is far from accurate, while our implementation for Atomic Counter delivers the correct result at a higher runtime. We can also apply this principle of “correctness vs speed” to our scalable counter, which performs even faster than our atomic counter, but at an “approximate” correctness, compared to the precise counter value in our Atomic Counter.

Names: Craig Li, Prerak Patel
NetIDs: craigli, pjp179
ilab Server used: kill.cs.rutgers.edu
CS416 - Operating Systems

Project 3 Report

Spreadsheet of Data used for Graphs:
https://docs.google.com/spreadsheets/d/1v3WFfaIgfca362Jstx4JSjGbsyXpv8Y54CFPTjftZRs/edit?usp=sharing

Testing Methodology

Data for each of our counters was taken as an average over 10 tests with the same number of threads (automated through a bash script). Each counter was tested with 1, 2, 4, 8, 16, 32, 64, and 128 threads, of which each test was repeated 10 times, and an average was taken. Graphs are included to show performance of each individual counter program (Figures 2-5), as well as the performance of all four overlaid onto a single chart (Figure 1).

Figure 1

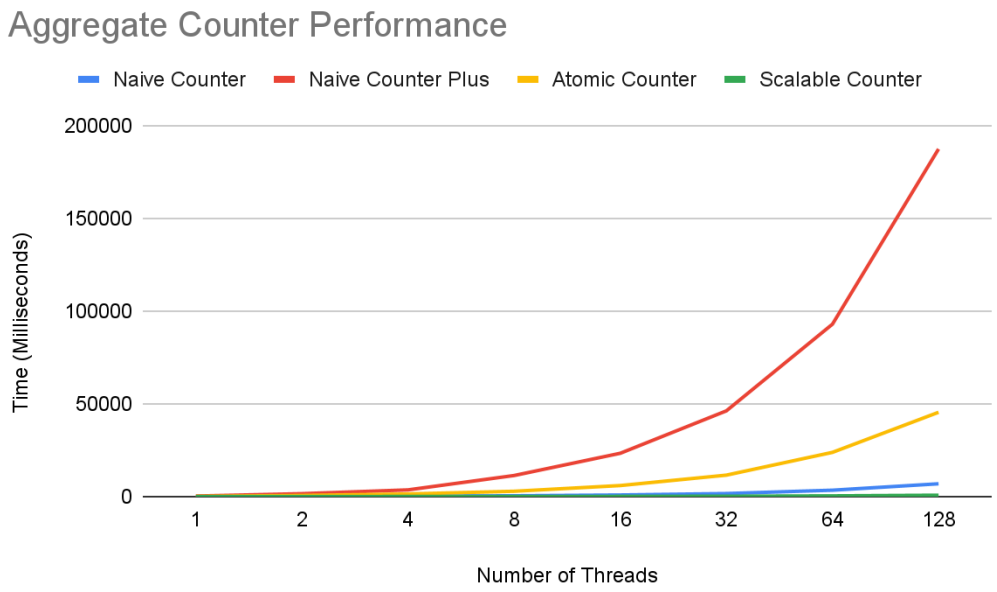
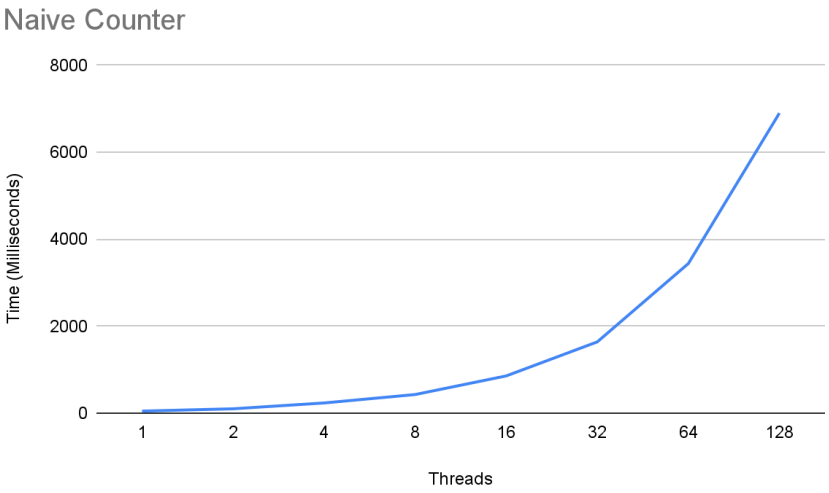


Figure 2



Names: Craig Li, Prerak Patel
NetIDs: craigli, pjp179
Project 3 Report
ilab Server used: kill.cs.rutgers.edu
CS416 - Operating Systems

Figure 3

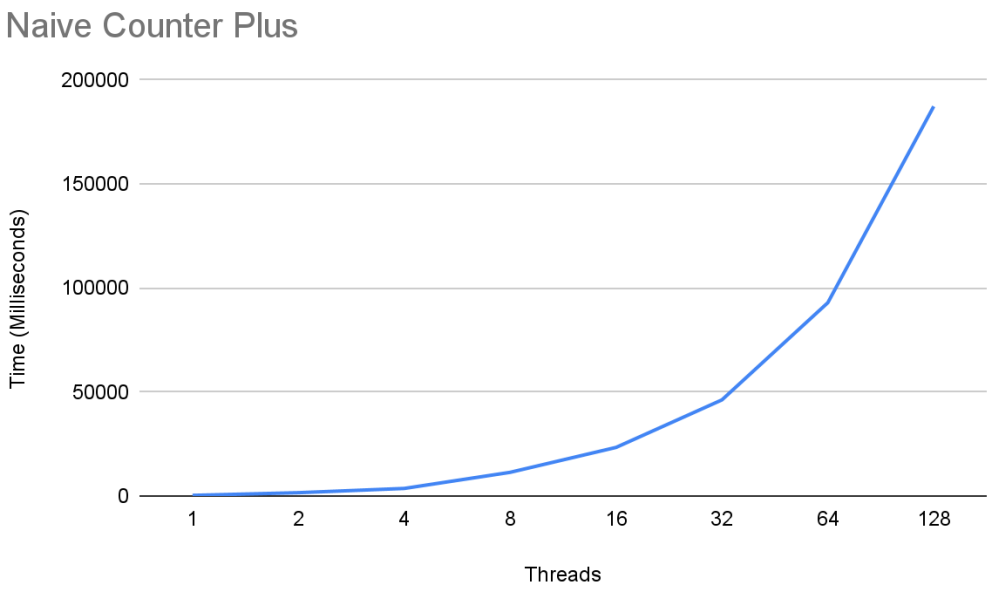
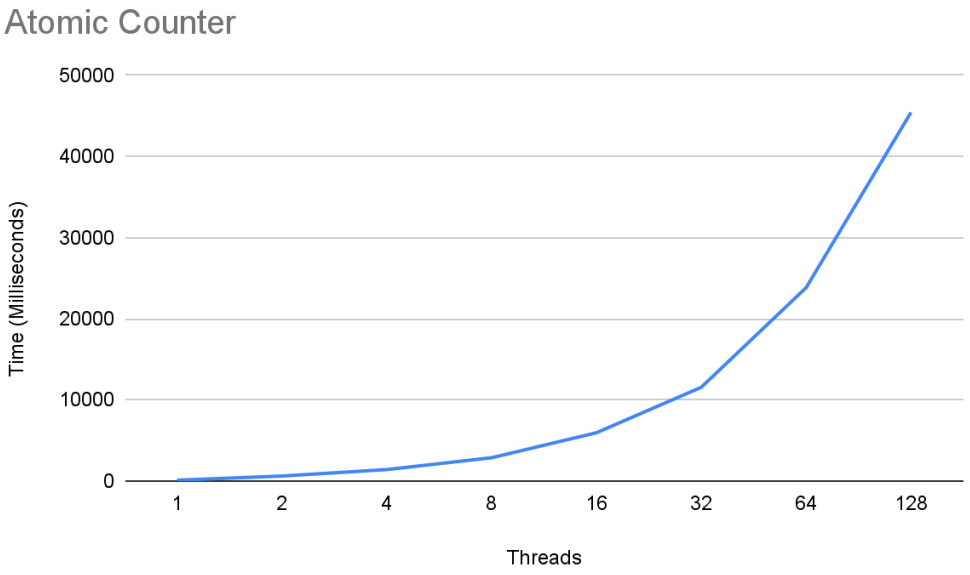


Figure 4



Names: Craig Li, Prerak Patel
NetIDs: craigli, pjp179
Project 3 Report
ilab Server used: kill.cs.rutgers.edu
CS416 - Operating Systems

Figure 5