Justin McBride, Warren Ferrell
CSCI 4446: Concurrent Programming
Pavol Cerny
17 February 2016
Modified: 16 April 2016

Final Semester Project Report

Motivation

Our other class, Object Oriented Analysis and Design, had a class project as well. Our group for that class made a web application, CoCal, that acts as a public events calendar. Multiple clients can add, modify, track, and delete events; these operations have to be managed through some central server and database, and so we believed it would be a good idea to implement our own backend infrastructure with this project. Because this app will be serving multiple clients at any single time, and clients can be modifying events at the same time as another client there is the possibility for many different race conditions and concurrency issues, which we solved with the skills that we garnered from this Concurrent Programming class.

Introduction

For this project we created a java program that stores and maintains the database for the CoCal web application (event and user information). All primary data structures for the storage of the calendar were created by the CoCal db team. In our search we found that database technologies already handle concurrent transactions. We believed that implementing this technology ourselves would be a good academic exercise and if done well may perform better than existing technologies that carry functionality not required for CoCal. Transactions that occur between the client and server involve the creation, modification, and removal of various entities.

Example

Two users are looking at the same event and notice something is wrong. They concurrently submit write requests to the database. The database must linearize the requests and report any conflicts.

Three users are using the app. User A is modifying event 62, changing the time. User B is going to delete the same event, while User C is currently loading into the application, and doesn't know about the event yet. Any of the actions can precede the other, as laid out below:

1. User A modifies the event; User B deletes the event; User C loads his calendar and the event isn't visible.
2. User A commits modifications to the event; User C loads his calendar and sees the modified event; User B deletes the event.
3. User B deletes the event; User A tries to commit his modifications to the event, but is warned that the event has been deleted (optionally giving him the choice to clone the event with his modifications, or abandon them); User C loads his calendar and doesn't see the deleted event.
4. User B deletes the event; User C loads his calendar and doesn't see the deleted event; User A fails to commit modifications to the deleted event and is warned appropriately.

5. User C loads the calendar and sees the original event; User A commits his modifications; User B deletes the event.
6. User C loads the calendar and sees the original event; User B deletes the event; User A fails to commit his changes and is warned appropriately.

Proposed Solution

Collections (users, calendars, groups, and events) will be represented in the server in two ways simultaneously:

1. The individual entities will be stored in RAM, as an object of its respective class.
2. The individual entities will be stored in files on disk. Each entity in a collection consists of a directory under the collection directory, wherein its attributes are files within that directory with their content corresponding the value of the attribute.
    - Users:
        - UUID
            - Account name
            - Password
            - Calendar ID
    - Calendars
        - Event ID list
            - EID
        - Owner
    - Group
        - Group Name
        - Calendar ID
        - Members
    - Events
        - Date
        - Name
        - Location
        - Owner ID

When a client submits a write request the database creates a thread for that write which requests a lock on a entity the client wishes to change. Lock requests on entities that have already been deleted will report to the user that their request could not be completed. Deleting specific entities will lock the entirety of the entity directory, so that other requests to it are blocked.

Quantitative evaluation

We wrote a companion test application that creates an arbitrarily large number of requests for the database to process. With these requests, many operations are queued up on either the same events or different events, and the time it takes to complete each request was measured. As well as being measured, we will ensure the results are correct. We created a variety of tests, where adding a plethora of events simultaneously thrashes upon the unique id counter of the events, or where trying to delete the same event from multiple clients simultaneously.

Implementation

GitHub: https://github.com/justinmcbride/CoCalDB

The project currently stores, loads, and edits files stored on the machine for persistent storage. As such, the project is mostly fleshed out and can perform all of the basic, required operations. Deviations from our proposed solution are denoted by a single *.

Concurrency in this database is accomplished as follows:

1. A request is received to perform an operation on the dataset
2. A thread task is submitted to a java executor*
    a. For comparison also tested spawning of a thread for each task.
3. The thread calls into our database to perform the operation
4. The collection data structure is modified
5. Any necessary locks are acquired
6. File operations are carried out
7. The thread dies and the request is completed

Entities (individual *Event*s, *Calendar*s, *User*s, and *Group*s*)* in the database are stored in a non-volatile manner via files in a tiered hierarchy depending on what they're representing, as laid out in the "Proposed Solution" section of the document. Small modifications were made, such as adding additional fields to the *Event,* object, and adding a reference solution, to refer from one object to another. We found that it was inefficient to concurrently modify file contents and much more efficient to create an individual file for each reference.

To prevent concurrent access to files on the database, the *Database* object manages a concurrent list for each *Collection* (all *Event*s, *Calendar*s, *User*s, and *Group*s), Each entity inherits from the Collection object and maintains a list of file objects which contain the value of said attribute and the filepath to that attribute's file. The underlying implementation of each collection list is dynamic, and can be changed at runtime. We have two types of management solutions: *LazyList*s, and *LockFreeList*s. Each of these is a list-based implementation of concurrent access protection, as learned through the semester.

When accessing an Entity through the database, the thread goes through the *Collection*, which performs the appropriate locking and unlocking (or not) to modify/read the *Collection* list. Then locks the Entity in the *Collection* if performing some sort of modification to allow access to the underlying data in an appropriate manner.

Testing Method Ideology

We predicted the performance variations we were likely see would come from thread count, physical memory implementation, thread management, and processor count. As such, we perform each of the following tests multiple times, and then average the running time durations:

Create **n** entities in the database.
Edit all **n** of those entities.
Delete all **n** of those entities.

Each of those operations is performed **m** times, with **x** number of threads, with **y** method of list synchronization, with **v** method of thread management, on a **z**-core processor.
As such, we are able to tune 5 different parameters to quantize the effects of each different factor. Every test was conducted at least 3 times with the min and max thrown out.

Testing Results and Discussion

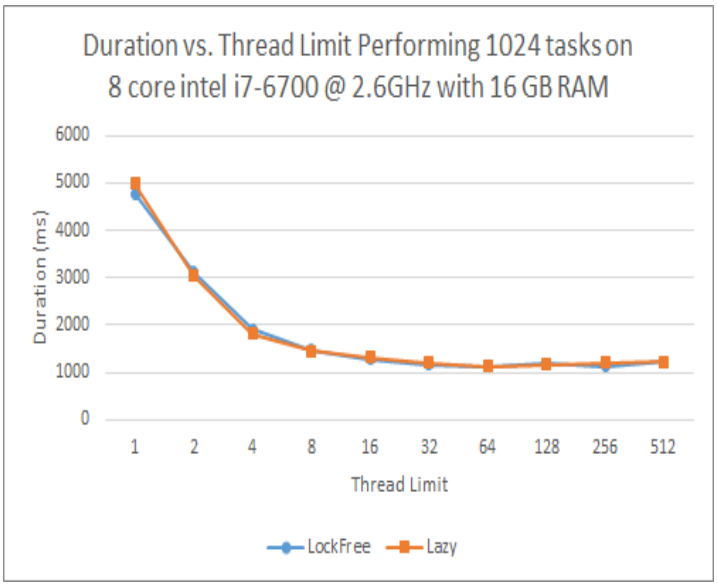

fig.1 Comparing Lock-Free and Lazy List implementation while spawning every thread. No significant difference and unreliable across different trials.

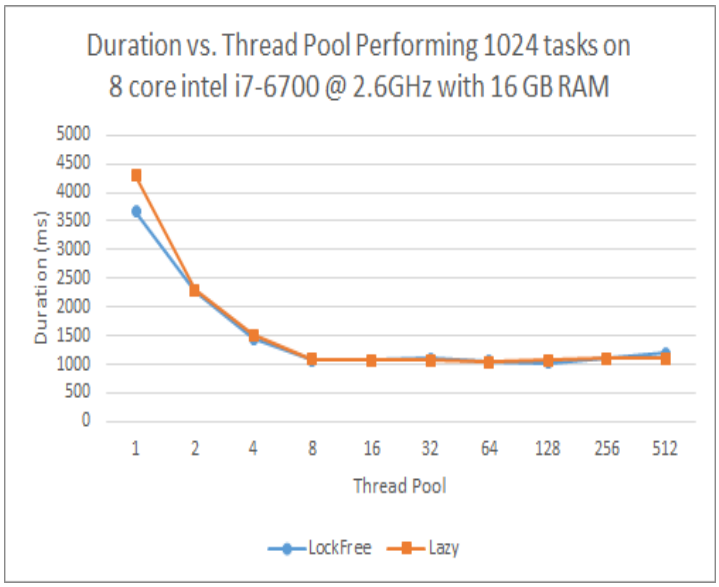

fig.2 Comparing Lock-Free vs. Lazy List implementation while submitting tasks to a thread pool. No significant difference and unreliable across different trials.

We saw very little performance gain when varying our list implementation on an 8 core processor (fig.1 and 2). We theorized that lock-free lists would perform better so we continued with that implementation for comparisons between the two thread management schemes. Thread pooling performed significantly better at lower total thread creation numbers (fig.3) but

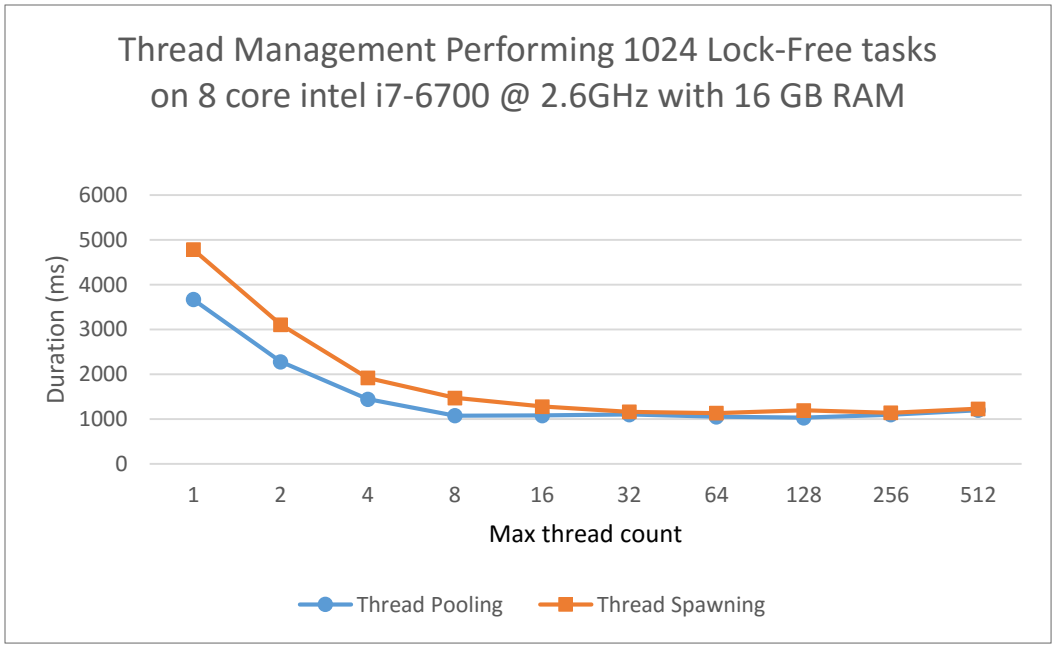

fig.3 Thread pooling vs Thread spawning, performing 1024 concurrent operations See very little difference over this range

did not appear significantly different with high thread numbers. To explore this difference, we chose 64 as a limit for total threads running and performed both management schemes with varying numbers of tasks performed (fig.4). We saw a significant increase in the performance of thread pooling as the number of tasks grew higher than those conducted in previous

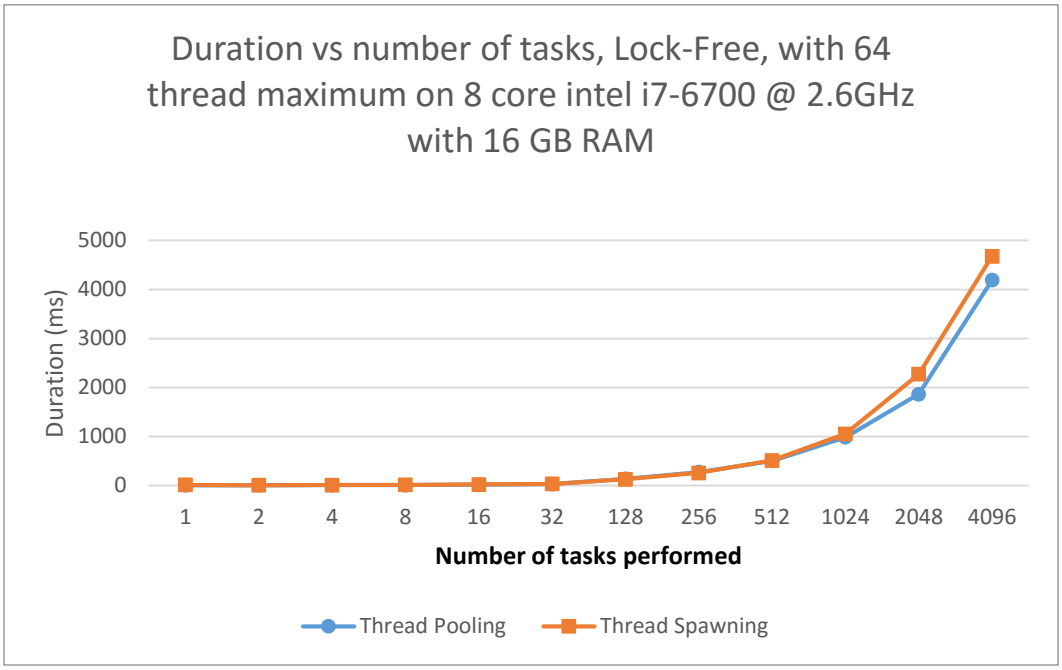

fig.4 Thread pooling vs. spawning across varying numbers of performed tasks. As the number of simultaneous tasks grows above 1024, thread pooling begins to perform significantly better.

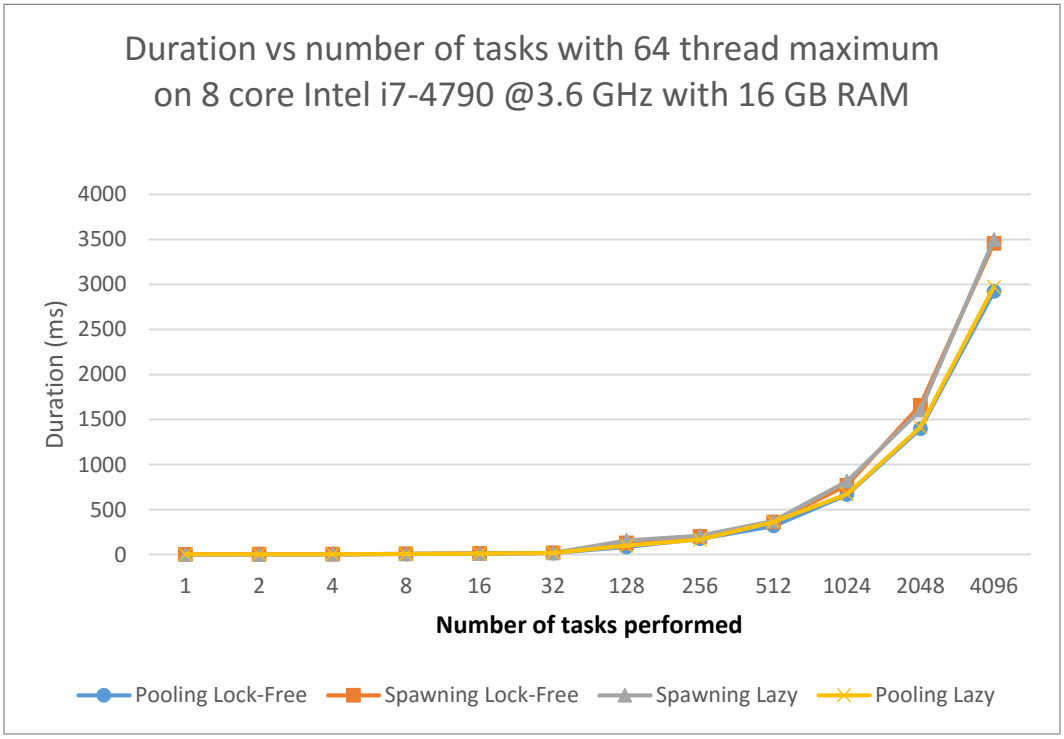

fig.5 Conglomerated results of above experiments on a different processor running Windows 7. Results match what was found above

experiments. Likely due to the accumulated overhead of re-initializing every individual thread. Repeating the above experiments on a similar machine setup produced the same qualitative result in a shorter amount of time largely explainable by the difference in processor speed (fig.5). The first results were conducted on a 2016 laptop while the later on a customized desktop.

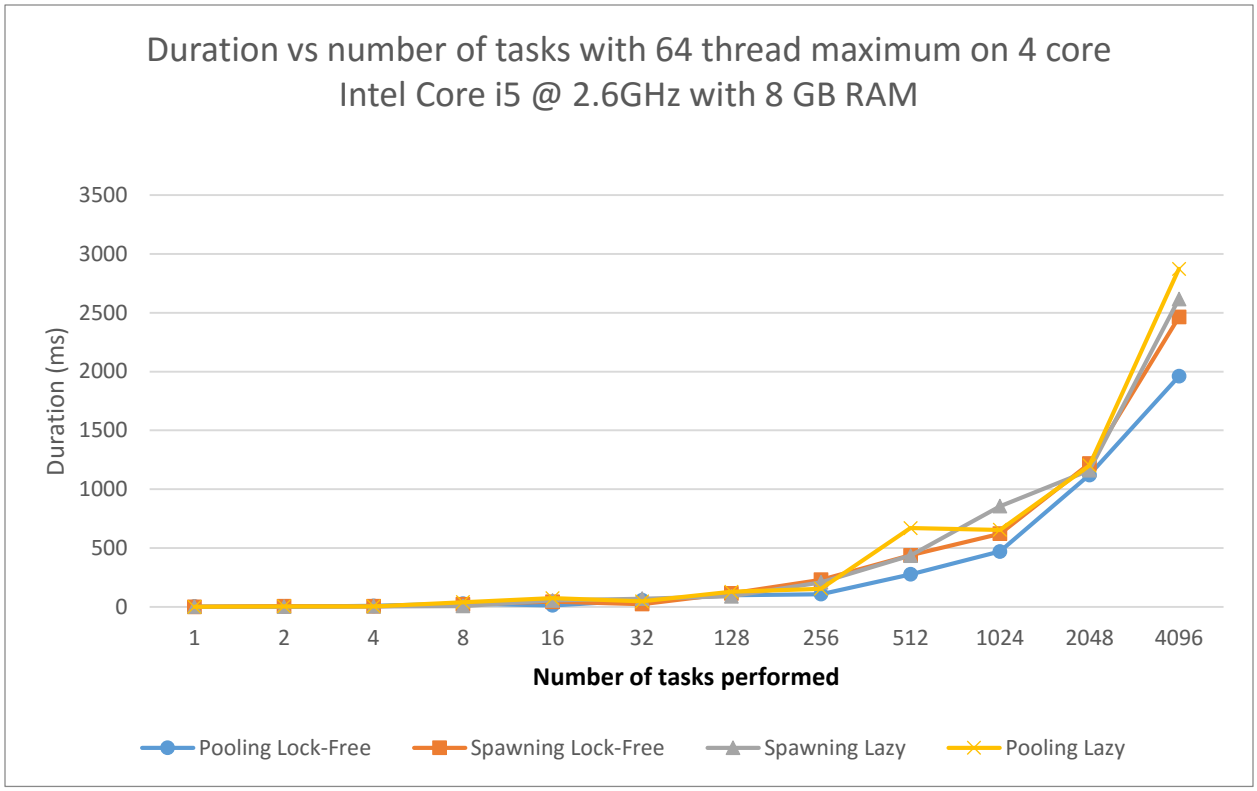

fig. 6 Conglomerated results from 4 core processor. See a greater distinction between Lock-Free and Lazy List implementation. Run in a MAX OS environment, contrary to the tests conducted in Windows 10 in the above experiments

The results from our 4 core processor (Laptop) test with 8 GB of memory vary quite significantly from the 8 core and can only be explained by the use of MAX OS instead of Windows. The overall speed was higher than both 8 core processors and the Lock-Free implementation showed a better performance than their Lazy counterparts. This result was so bizarre that we repeated the tests multiple times but always produced the same trend. Memory management, file management, and scheduling are all handled slightly differently between the two Operating Systems. Running the same tests on a Windows 10 4 core Laptop with 8GB RAM performing more than 512 concurrent tasks caused errors when windows memory management leaked memory. Both Windows 8-core runs ran into problems running more than 4096 tasks due to file permissions errors thrown by Windows (possibly a safety feature within windows preventing extremely high directory access as the test only had 64 concurrent tasks running at each point). Future experiments should explore usage under other operating systems and alternative non-volatile storage options outside the OS's default file storage mechanism. Overall we thoroughly enjoyed this project, the challenges it presented and believe it was an excellent learning opportunity.