Report

UNSW SYDNEY

Assignment 1

Implementing Concurrent Operation
of a Data Set Based on Read-Write Lock

Student: Jinghan Wang

Student code: z5286124

Course: COMP3151

Term: T2 2023

Due date: 2023-07-17

Contents

1 Introduction

This project mainly utilizes read-and-write locks as the main body to ensure that only one process is allowed to run if and when the data needs to be modified, and multiple processes can read the data if there are no writes. In addition, signalling is used to ensure the integrity of the data structure.

2 Solution

First, a separate read/write lock is given to each piece of data, allowing member() and print_sorted() to read the data freely and ensuring that, at the exact moment, no two processes are allowed to modify the data simultaneously.

Second, the number of signals is set equal to the maximum capacity of the data structure so that when the maximum capacity is reached, the insertion process is blocked to know that data has been deleted. It guarantees the data structure's integrity and ensures that the program it wants to insert will not be terminated directly.

2.1 Insert

Insert() chooses to traverse the list to ensure the inserted data can be placed correctly. Our primary concern is to find cases where the value at a location is greater than the inserted value and insert it. Also, due to the deletion mechanism, when the program finds an empty position, it fixes a pointer i to record the position and continues searching backward.

- If it is an empty position, it continues backward.
- If it encounters a value in position j smaller than the inserted value, it moves the value in position j to position i. Position j is set to empty, and i points to the next position. It aims to avoid the deadlock problem caused by the need to request a read lock forward after finding a value greater than the inserted value. However, as a cleanup method, when we insert a maximum value, the program will fill the vacancy to completion.
- If equal, then the data structure is not changed in any way and ends
- If it is greater than or all the empty spaces after i have been checked are empty, then n is placed at position i

Regardless of the best case or worst case, it has a time complexity of $O(N)$, and its rate depends on the position to be inserted as well as the current distribution of the data. In most cases, it is necessary to traverse the entire list to find the appropriate position. For concurrency issues, since read locks cannot be directly upgraded to write locks, write locks are applied to insert queries to ensure that there are no concurrency issues that might be caused by requesting a write lock after releasing a read lock.

At the same time, all processes are requested from the beginning to the end to ensure that when an insert process is making changes, another insert process needs to wait until the insert is completed before it can query the data.

Moreover, Release a location as soon as it is finished modifying it, reducing the time spent unnecessarily occupying the critical section.

2.2 Delete

There are two ways to realize the implementation of delete().

- The first one is more straightforward, use dichotomy to find the corresponding position; when the empty position is encountered, search to the left only to find the position that has a value to compare; if the left side is empty, then set a start as the intermediate value, and proceed to the next iteration.

  The reason for choosing this way is because, due to the existence of the mechanism of cleanup at the same time as the insertion of the insertion is from left to right as such, in the middle of the value of the left side of the greater probability of finding a comparable value. When the program is deleted, the less, the easier to find the value, so added, when the program successfully executes the capacity of half of the deletion operation, the cleanup operation.

- The second way is more complicated; the reason for trying is that the above mechanism may have the possibility that the original value is outside the start and end range due to insert. Therefore, when start = end but still not found in the case, his implementation is trying to move around to find the correct position. Although no problem is found in the Java test segment, there is a deadlock problem in the Promela test.

In the best case, i.e. there is no null value in the data structure, it has a time complexity of $O(logN)$ for bisection. However, if there is no insertion in the data structure, it is the worst case with $O(N)$ time complexity. Thus, the content of values in a data structure determines the efficiency of its operations.

For concurrency, the program applies a write lock to all reads of a value to minimize possible concurrency problems.

2.3 Member

Member() uses the same implementation scheme as the first method of delete, simply changing the write lock to a read lock; as long as there is no need for insert and delete, they will not be blocked.

It has the same time complexity as delete(), which is closer to $O(logN)$ when the number of values is higher and conversely closer to $O(N)$.

2.4 Print_Sorted

Print_Sorted() traverses the entire list using a read lock and will not block when there is no write lock. Its time complexity is $O(N)$.

2.5 Cleanup

The program cleanup is included in delete(), which uses Java's AtomicInteger to ensure atomicity in program calls and checks. At the same time, implementing the insert allows him to call only the most considerable int value; Avoid deadlocks by calling a new thread to do the insert; It can fill the gaps in the data structure after deleting the most considerable int value to ensure the program's integrity.

Its time complexity is $O(N)$, which must traverse the entire list.

3 Safety and Liveness

3.1 Read-Write Lock

Read and write locks are the data structure's central part. It satisfies the mutual exclusivity in security and the no-starvation in the active specification.

For mutually Exclusive, at any given time, only one thread can hold a write lock, or one or more threads can hold a read lock. In other words, write operations must be mutually exclusive; read operations can be performed concurrently but not concurrently with write operations.

For no starvation, if a thread requests a lock, then it should eventually be able to acquire the lock.

3.2 Semaphore

The setting of semaphores ensures finiteness in the safety property and no starvation in the liveness property.

For finiteness, the value of the signal quantity does not exceed the maximum capacity of the data structure, which ensures that the system's data structure is not overused and that no data loss occurs. For starvation-free or final entry, all insertion processes can complete the insertion if they have a position. Every thread has the opportunity to acquire a signal volume.

3.3 Program

The security properties of mutual exclusivity and inorganic starvation in the liveness property are met for the program.

For mutual exclusivity, insert ensures that if multiple write locks need to be requested, the order is from left to right, and only the process holding the write lock can modify a range of data; if another process wants to modify the same range, it is constrained by this program to wait for the process to finish releasing the write lock before it can request it.

For starvation-free or final entry, when a program requests a write lock, it will eventually acquire it.

4 Java Implementation

For implementing the program, first, the data part uses **ArrayList** to store non-negative integers and -1 as an empty position.

The concurrency part mainly uses **ReentrantReadWriteLock** and **ArrayList** to manage the read/write lock arrays, **Semaphore** to manage the space left, **AtomicInteger** manages the atomic operation of cleanup to complete. They are all private properties encapsulated in **ConcurrencyDataStructure**, leaving only 6 API's ensure that external data do not modify the data structure

- **public ConcurrencyDataStructure(int n)**: Initialize
- **public void insert(int n)**: Insert
- **public void delete1(int n)**: and **public void delete(int n)**: Delete
- **public boolean member(int n)**: Find member exists
- **public void print_sort()**: Print line

For the Thread part, there are four java classes, which all implement Runnable; the initialization will be **ConcurrencyDataStructure** and the value to be operated will be passed in and recorded, and then the corresponding code will be called in **public void run()** to complete the corresponding operation and print the corresponding state.

5 Optimization Possibilities

For this program, since the operations of deletion and finding are for the whole data set, the efficiency of deletion and finding may be optimized if the program can set a current end position.

Also, in my conception, if delete operation first uses a read lock to find the approximate location. Then a write lock for further searching makes less use of write locks, it seems to reduce the number of instances where searching and printing are blocked.