CSC3021 Concurrent Programming – Assignment 3

Question 1 – Christmas Tree Shopping

Shared Variables

```
int logger = 0;
int customerWaiting = 0;

Semaphore noOfLoggers = 3;
Semaphore treesLeft = 20;
Semaphore customerInStore = 0;
Semaphore loggerInStore = 1;
Seamphore noEntry = 0;
```

Process Merchant	**Process Customer [1..C]**	**Process Logger[1..L]**

while(true){
down(treeLeft);
down(customerInStore);
}
 	
End Merchant
	while(true){
	if(logger == 0){
	enter();
	up(customerInStore);
	exit();
	}
	else{
	customerWaiting++
	down(noEntry)
	}
	
	}
	
	End Customer
		while(true){
		logger++;
		down(noOfLoggers);
		down(loggerInStore);
		enter();
		up(treesLeft);
		up(treesLeft);
		up(noOfLoggers);
		exit();
		logger--;
		if(logger == 0){
		if(customerWaiting >0){
		while(customerWaiting !=0){
		customerWaiting--
		up(noEntry)
		}
		
		}
		}
		}
		
		End Logger

Question 2 – Lock-Free Hash Map

Linearization Points of get(), add() and remove() methods

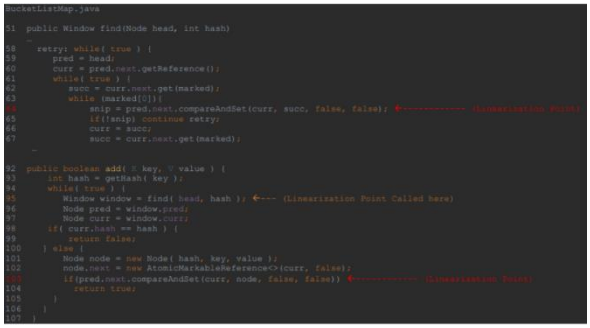
get() Linearization Point

```
BucketListMap.java

77   public V get( K key ) {
78     int hash = getHash( key );
79     Node curr = this.head;
80     while( curr.hash < hash )
81       curr = curr.next.getReference();
82     return ( curr.hash == hash ) ? curr.value : null; <---------- (Linearization Point)
83   }
```

For the get() method, the linearization point is on line 82. Line 82 is the linearization point in the get() method as this is the point where a value is returned from this method and other method calls will start to see the effect of this method

add() Linearization Points



There are multiple linearization points in the add method, one can be found in the add method on line 103 and the other is found line 64 in the find method which is called on line 95. The first linearization point is when the find method is called and compareAndSet is called in find method. This is a linearization point as the value of curr has been updated with succ and this update will be seen by other called methods. The next linearization point can be found on line 103. The reason this is a linearization is similar to the first linearization point, we update the value of Node curr with the value of Node node and this update will be seen by other currently executing methods.

remove() Linearization Points

Like the add() method, there are multiple linearization points in the remove() method. The linearization points occur when the find method is called on line 134 and occurs in the find method on line 64 and the next linearization point is on line 142. The first linearization point is the same as the first linearization point in add() where the value of Node curr is updated with the value of Node succ and this update is seen by other currently running methods. The next linearization point happens when we unlink the Node succ on line 142. We mark this node to show that it is unlinked. Other currently running methods will no longer be able to access this node which shows that this is the linearization point.

Charts to analysis throughput of different HashMap implementations

Impact of segments on SegmentedHashMap Throughput

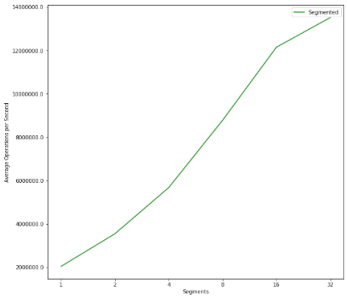


Figure 1

The graph in Figure 1 was created with the following settings:- java Driver 8 8192 <num_segments> 8192 18 1000 10 segmented. As we can see with the chart in Figure 1, as the number of segments goes up, as does the average number of operations. This tells us the we can get increased throughput by introducing more segments to the SegmentHashMap. We get improved throughput due to splitting the hashmap into more manageable segments and through synchronization, we can also run more manageable segments on multiple threads, which helps improve throughput

Impact of threads on a HashMap's Throughput

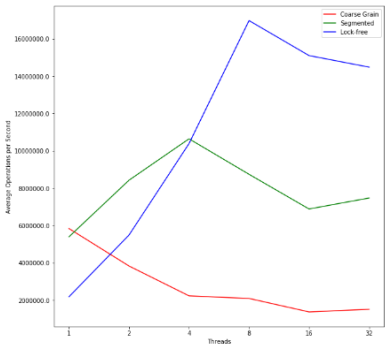


Figure 2

The graph in Figure 2 was made with the following settings: -java Driver <num_threads> 8192 8 8192 18 1000 10 <type>. As we can see threads have a massive impact on throughput, both positively and negatively. For coarse grain, we can see that throughput gets worse as we increase the number of threads. This is due to the fact that all coarse grains methods are locked when one thread enters, this leads to bottleneck which cause throughput to get worse as the number of threads increases.

For Segmented, we can tell that initially it seems to benefit from the additional threads, but we hit are limit at 4 threads and throughput gets worse as we add more threads. This could be a problem with using locks in to synchronize each segment, this is beneficial with less thread, but as we add more and more thread, the bottleneck becomes more apparent since threads could potentially be stopping and waiting to operate on the same segment which leads to lower throughput

The lock-free implementation benefits the most from having more threads. Since this implementation doesn't have to wait for a method to open up, it can start executing as soon as its ready. This allows for increased throughput.

Impact of Get Operation on a HashMap's Throughput

The graph is figure 3 was created with the following settings:- java Driver <num_threads> 8192 8 8192 <get_ops> 1000 10 <type>. As we can see from the graph in figure 3, the percentage of get operations has a varying affect on each of the 3 HashMap implementations. For coarse grain we can set that initially there is a gain on throughput when using more get operations but as that increases past 75% we see a decline in throughput. The reason for the decline is due to it operating on a single thread which means as it gets more get operations, it will eventually be bottlenecked as it won't be able to perform as many get operations.

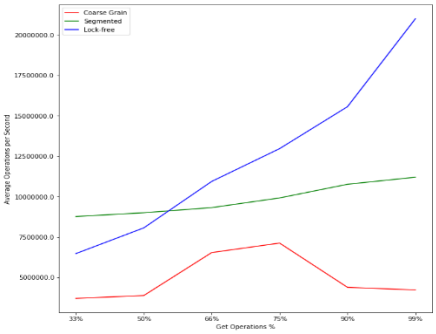


Figure 3

For segmented, we can that there is a very steady increase between the number average operations and throughput and its initial start off at the highest throughput with 33% gets. This tells us that the number of get operations does have an affect on throughput for segmented but its very minimal overall. This is due to the way the synchronisation is implemented as we do have to stop on every segment which is the same and when we do more gets, we are bound to get more segments which are the same.

Lock free has best throughput off the free and their significant gains made each time the get operation percentage is increases. This is due to the lock free nature of the hashmap. Since there is no need to wait for a lock to unlock like the other two implementations, lock free can do as many gets as possible which allows it to go through the whole process quicker leading to better throughput.

Question 3 – Locks Galore

1. A SELECT query would have to make a read on the index of the record that has been selected by the SELECT query and a read to a record that has been selected

by the SELECT query. UPDATE would need to read the current value of the record that needs updated and write the new value to the record, it should not need to access the index. INSERT will need make a write access to the new index and record it will be adding, it will not need to have read or write access to this index or record.

2. There exists a data race that could occur between the SUM transaction and TRANSFER transaction. In the TRANSFER transaction, the ‘amount’ column is given write access, so the amount can be updated and the SUM transaction reads from the ‘amount’ column. This could lead to a data race if executed concurrently as an account could have it value added in twice which would lead to an incorrect value because of an account could be read with it original amount and the updated amount. Deadlock potential does exist between TRANSFER and SUM since both need access to the same shared resource, ‘amount’. One of the transactions could keep a hold of one records which could lead to one of the processes being deadlocked
3. There should be no data race condition between the CREATE transaction and TRANSFER transaction. Since the CREATE transaction inserts a new account into account table, the TRANSFER transaction will not have access to this account yet and since TRANSFER would only work between two existing account. For the reason above, we should also not encounter in deadlock with the CREATE and TRANSFER transaction as they are not competing over any shared resources.
4. There does exist a race condition between SUM and CREATE. As a new account is being created, a new amount is also being added. There is a chance since this is writing to a new ‘amount’ record, SUM might not get this value and calculate the incorrect value. There is no deadlock potential between SUM and CREATE transaction. Even though both have access to the same shared resource ‘amount’, CREATE is adding a new amount which SUM wouldn’t have access to until it has been added which means that there is no potential for a deadlock
5. There could exist data races and deadlock with multiple TRANSFER transactions with at least one account being the same. Since there are record level locks, there could be deadlock as multiple TRANSFER transaction may need access to the same account and not get it. A data race could exist if there is multiple TRANSFER transaction with the same account multiple times.
6. For the CREATE transaction, we will need to create a table level lock, lock it and then unlock it to maintain total order among locks. For the TRANSFER transaction, we create two record locks between the 2 account records, lock both and when the transaction is finished, unlock both and we will maintain total order among locks for the TRANSFER transaction. Finally, for the SUM transaction, we create a table level lock and lock and while we go through each record to sum, we obtain a record lock and lock. When we get to the end of the transaction, we unlock all the record level lock, then table lock to main total order between locks.

Improving Performance of solution

By changing table level lock from a readLock to a writeLock in SelectForUpdate we can improve performance. This is because multiple threads can obtain the same readLock which can lead to a bottle where only one thread can acquire a writeLock at one time. Because of this we get performance deficits when using readLock for the table when we add more threads. By using a writeLock, we can stop this deficit to improve performance.

I have also added a check to see if the size of all the records to be locked I less than half the size of all records in the table. If it is less then we use record locks, if not we continue with table level lock. Since we are assigning less locks we can improve throughput