**CSC3021 Concurrent Programming – Assignment 3**

**Question 1 – Christmas Tree Shopping**

|  |  |  |
| --- | --- | --- |
| **Shared Variables**  int logger = 0;  Semaphore noOfLoggers = 3;  Semaphore treesLeft = 20;  Semaphore customerInStore = 0;  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{  down(noEntry)  }  }  End Customer | while(true){  logger++;  down(noOfLoggers);  down(loggerInStore);  enter();  up(treesLeft);  up(treesLeft);  up(noOfLoggers);  exit();  logger--;  if(logger == 0){  up(noEntry)  }  }  End Logger |

**Question 2 – Lock-Free Hash Map**

1. **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; 🡨------------  
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**

BucketListMap.java

92 public boolean add( K key, V value ) {  
93 int hash = getHash( key );  
94 while( true ) {  
95 Window window = find( head, hash );  
96 Node pred = window.pred;  
97 Node curr = window.curr;  
98 if( curr.hash == hash ) { 🡨------------  
99 return false;  
100 } else {  
101 Node node = new Node( hash, key, value );  
102 node.next = new AtomicMarkableReference<>(curr, false);  
103 if(pred.next.compareAndSet(curr, node, false, false)) 🡨------------  
104 return true;  
105 }  
106 }  
107 }

There are multiple linearization point in the add() method. They are at line 98 and line 103. At the first linearization point, we find that the hash of Node curr is equal to hash and return false and any currently running methods will see that the effect of this method. At the second linearization point, we find out that hash is not equal to curr.hash and make a new node based on the key, value and hash and then set it to node after pred. Since we are adding another node to the linked list, other methods will be visible to the other currently running methods. This method has multiple linearization points since there multiple exit points which depending on the execution will lead to different results

**remove() Linearization Points**

BucketListMap.java

130 public boolean remove( K key ) {  
131 int hash = getHash( key );  
132 boolean snip;  
133 while( true ) {  
134 Window window = find( head, hash );  
135 Node pred = window.pred;  
136 Node curr = window.curr;  
137 if( curr.hash != hash ) 🡨------------  
138 return false;  
139 else {  
140 // Unlink node  
141 Node succ = curr.next.getReference();  
142 snip = curr.next.attemptMark(succ, true);  
143 if(!snip)  
144 continue;  
145 pred.next.compareAndSet(curr, succ, false, false); 🡨------------   
146 return true;  
147 }  
148 }  
149 }

Like the add() method, there are multiple linearization points in the remove() method. The linearization points happen on line 137 and 145. The first one occur when curr.hash does not equal hash and false is returned. This is a linearization point as it signals to the other methods that a variable will be changed. The next point occurs when we unlink the node from the linked list. This is because the current linked list is being changed and this change will be visible to all other executing methods.