**Assignment 1 - Design Document**

**Construction of Circular Double Linked List Data Structure**

* Every circular double linked list got to have a "head" pointer that points to the first element of the linked list, the list head is initially null as there are no elements in the list

**private** **volatile** Element listHead = **null**;

* The constructor of the linked list creates a CDLList object with just one node which is the head of the list and with its previous and the next node pointing to itself as there are no other nodes in the linked list.
* The method **public Element head()** returns a reference to the head of the circular double linked list.
* The CDLList class has an inner class called the **Element** which actually indicates the type of the elements in the circular double linked list, every node constituting the circular double linked list will be of type Element, consisting of 3 fields a next pointer, a previous pointer and a node value.
* The method public Cursor reader() returns a cursor to a specific element in the cdllist, the cursor object is actually a placeholder where we can store the location of a specific element of the circular double linked list and traverse the entire linked list using the circular nature of the list.
* The CDLList class has an inner class with the name Cursor, which is actually used to provide the functionality of the cursor literally, it has methods that return the current position of the cursor object and move the cursor to an object succeeding its current position or to an object preceding its current position.
* The **public Writer writer ()** method in the cursor class returns a writer object relative to the cursor position, so once you instantiate a writer object using the cursor object, the position of the writer object is fixed and will never change and all the insertion operations that is **insertBefore** and **insertAfter** will take place relative to the current position of the writer. The Writer class has methods for inserting new nodes relative to the current writer position.

**Inheritance of Inner Classes While Locking**

Some crucial points concerning **Inner Classes** in Java -:

* Inner classes are members of the enclosing class and hence has access to the private fields and methods of the enclosing class’s instance as well.
* As Inner classes are **Non-Static** classes they are tied or associated not to the class as a whole but to each and every individual instance of the class. Hence in the assignment given the inner classes **Cursor**, **Element** and **Writer** are all tied to individual **CDLList<T>** instances.
* As the instances of the inner class can exist only within the instances of the outer class, each Cursor, Writer and Element object exists within an instance of the CDLList<T> class.
* An invocation to the reader method such as

“***CDLList<String>.Cursor*** ***c = f.reader(f.head())***” will invoke the reader method of the CDLList object “***f***” and if we take a look at the code of the reader method which is as follows -:

***public*** *Cursor reader(Element from) {*

*Cursor newCursor =* ***new*** *Cursor();*

*new Cursor.setCursorPosition(from);*

***return*** *newCursor;*

*}*

This is equivalent to the following code with an explicit “**this**” pointer notation.

***public*** *Cursor reader(Element from) {*

*Cursor newCursor =* ***this****.****new*** *Cursor();*

*new Cursor.setCursorPosition(from);*

***return*** *newCursor;*

*}*

Which indicates that this Cursor is tied to the CDLList object “**f**”. Hence in the code it is implicitly tied to “f” without the use of this pointer invocation.

* We make use of all the above facts in designing our Locking protocol as that requires extensive inner class inheritance.

**Coarse Grain Locking**

As the name coarse grained locking means, it is the most crude way to lock objects and get the task accomplished, multiple threads execute essentially sequentially when the shared object they are working with is locked in a coarse fashion. The implementation details of the coarse grain locking protocol is as follows -:

* In order to implement the coarse grain locking protocol we extend the CDLList<T> generic class that represents our data structure class essentially, so all the functionality and private variables of the parent class do actually form members of the child class which is CDLCoarse<T>.
* Each instance of the CDLCoarse<T> class will have an object called “lockObject” which as the name suggests will be used for synchronizing actions and communications among several threads.
* The only thing we need to worry about in the coarse grained locking is how to insert values and nodes in the linked list in a synchronous fashion and that is accomplished using a single instance of the “Object” class per circular double linked list object.
* Essentially a **synchronized(lockObject)** block of code will serve the purpose which allows different methods to access the CDLList object in a synchronous fashion

**Fine Grained Locking**

In the fine grained locking technique we get to a more individualistic form of locking where we lock access to individual elements of the circular double linked list -:

* To insert an element before a node where the writer currently points which is actually the position of the cursor at which the writer was instantiated, we use the following protocol which is lock the previous node, then lock the current node and then insert the element, so that no other thread is able to gain access to the previous node and then modify it as not locking the previous node might lead to unnoticed inconsistencies.
* To insert an element after a node where the writer currently points which is actually the position of the cursor at which the writer was instantiated, we use the following protocol to prevent inconsistencies, lock the current node at which the writer points, then lock the node that succeeds the current node at which the writer points, so that no other thread is able to gain access to the next node and modify as not locking the next node might lead to unnoticed inconsistencies.
* There are certain conditions that might lead to deadlock in the multi-threaded circular doubly linked list data structure, for instance if there are n threads and n nodes in the circular double linked list and n threads try to insertAfter(T val) or insertBefore(T val), then in that condition each thread either tries to acquire a lock on the current node and the next node or the previous node and the current node and that might lead to cyclic dependency in the graph, which is actually one of the conditions for deadlock. To remedy this situation we use a “dummy” variable which is not visible to the client, but is internally used to prevent the deadlocking conditions, this dummy variable can be a floating dummy variable as long as no thread can have access to it.

**Read/Write Locking**

There are essentially 2 major conditions that govern the phenomenon of Read/Write locking, the context in which we apply Read/Write locking in our systems is when the number of Read requests outnumber the Write requests for a resource, the conditions are as follows -:

**Read Access -:** We give access to all those threads that want to just read off the value of the resource, let us say if a text file is considered as a resource, then multiple threads can gain access to the file as long as they are reading the file, but in case there are already write requests to the resource or in case there is some thread writing to the file currently, all the threads that want to read the file must wait before the writing thread finishes its task and notifies all the threads that are waiting on that resource. So, the condition for providing the read access to multiple threads would be something like this: ***(numberOfWriteRequest == 0 && numberOfWriters == 0).***

**Write Access -:** We give access to a thread that wants to write to a resource such as a file or a memory location as long as there are no threads writing to the thread or reading the thread, that is the following condition: ***(numberOfReaders == 0 && numberOfWriters == 0).*** Even if there are a number of read request while we grant the access to a writing thread, we don’t care about them because it is only when the write thread notifies all the threads that it releases the lock on the object or the critical section it was working in. Hence those read requests can accumulate but they have to wait.

**Lock Fairness -:** if there are multiple readers that are holding the lock to an object and there is a writer thread that wants access to that lock for writing the data in the critical section, then no new reading threads will be allowed to take the lock even if they don’t cause problems this condition needs to be there that is we need to up-prioritize the write locks because they may starve if this isn’t done.

**CDLCoarseRW class implementation**

In implementing the CDLCoarseRW class the following steps are performed -:

1. Each **CDLCoarseRW** object will have an instance of the **ReadWriteLocks** class as its member, this will serve as the one read/write lock per list data structure.
2. The CDLCoarseRW constructer accepts a value of type “T” which is the template parameter and then initializes the head of the linked list.
3. The “**head()**” method of the CDLCoarseRW class which returns the reference to the head of the circular double linked list, is actually a read request that a thread might make to the CDLCoarseRW instance, hence we need to synchronize it. Well, we should synchronize the access to the head () method solely because there might be a thread that can just access the head of the linked list and display its elements, but this can be dangerous because there might be some other thread that can be writing to the linked list and hence in case if there is such a thread, then the read thread must wait before the writing thread is finished.
4. The same is the case with the **reader** () method of the thread which returns a cursor object to a specified element in the circular double linked list. This method also needs to be synchronized because a reader thread might access the circular double linked list using the reader method while some other thread is writing to the linked list.

**CDLCoarseRW class implementation**

In implementing the CDLListFineRW calls the following steps are performed-:

1. We make an instance of the ReadWrite class within the Element class whose elements actually form the elements of the circular double linked list.
2. As with the fine grain locking protocol for inserting the node after and inserting the node before a node, we perform the same steps using the read write lock objects inside the individual elements of the circular double linked list.