Sneha Banerjee
snehaban@buffalo.edu

CSE505: Assignment 1 - Concurrency Control

The Element class has the following protected variables:

```
public class Element
{
        protected T data;
        protected Element next;
        protected Element prev;
        protected boolean isLocked;
        protected RWLock rwLock;
}
```

isLocked	To check if the Element has already been locked by another Thread
rwLock	Used only in CDLListFineRW class so that each Element can be locked or unlocked individually using the methods of the RWLock class.

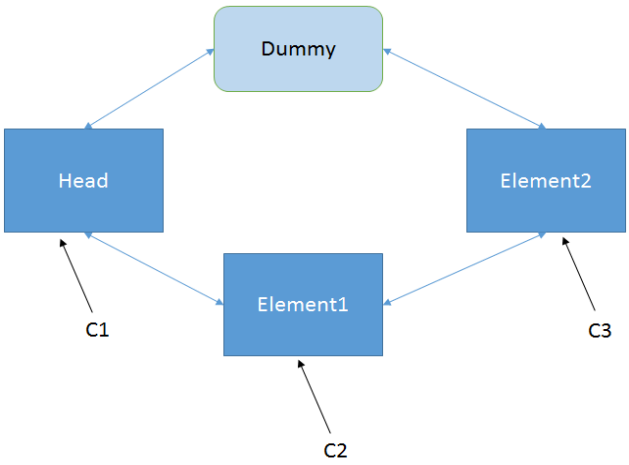
Fine-grained locking:

CDLListFine uses fine-grained locking for its Cursor and Writer operations. Any read or write operation requires to lock only the part (one or two Element) of the list on which it is working so that multiple threads can operate simultaneously on different parts of the list. This could lead to deadlock or race conditions if multiple threads try to update the same Element.

Deadlock avoidance:

Deadlocks can be avoided if locks are always acquired in a specific order, in this case from left to right. In a doubly circular linked list, a dummy element can prevent deadlocks. The dummy element is before the head element, but is not accessible externally to the user, so no cursor ever points to it. The Writer functions need to acquire both locks in the given order before modifying the list.

- *insertBefore(val)* - First acquire lock on previous Element, then on the current Element.
- *insertAfter(val)* - First acquire lock on current Element, then on the next Element.
- *delete()* - First acquire lock on previous Element, then on the next Element.



- Consider a list with 2 elements and a head.
- Add a dummy element just before the head, with its previous pointing to the tail and next pointing to the head of the list.
- 3 cursors C1, C2, C3 point to the head, element1 and element2 respectively.
- All 3 cursors perform ***insertAfter()***
- C1.writer requires locks on the head first and then on Element1.
- C2.writer requires locks on Element1 first and then on Element2
- C3.writer requires locks on Element2 first and then on the dummy.

- Without the dummy element, there would be a circular dependency on the head because both C1 and C3 would try to acquire locks on it, while holding another lock, leading to a deadlock.
- Since the dummy node is not accessible to any cursor, only one writer (Element2 in this case) would need to acquire a lock on it, thus breaking the circular dependency and avoiding deadlocks.
- In the scenario when all 3 cursors perform ***insertBefore()*** operation, only C1 would need to acquire a lock on the dummy. C3 would require locks on its previous (Element1) and current (Element2). Thus the circular dependency is broken and no deadlocks occur.
- Similarly in case of a ***delete()*** operation, there is no circular dependency.

Prevention of Race condition:

a. Lock on any Element prevents modification of its protected variables (its data, next and previous pointers, isLocked flag and the RWLock lock) by other threads. Once locked, an Element can only be updated by the locking Thread and once the lock is released, the updates become visible to all other threads.
b. The Writer operations of *insertBefore()*, *insertAfter()* and *delete()* cannot update the list until they obtain locks (Synchronized block) on both the Elements. So there is no shared data between the threads, preventing race condition.
c. The Cursor operations of *current()*, *next()* and *previous()* also need to lock on the current Element so that it doesn't read any inconsistent data while some other thread is updating it. Hence, race condition is prevented.

R/W Lock:

The RWLock maintains a *Concurrent Read and Exclusive Write* state of each Element.

```
public class Element {
	private volatile int readers; // current readers
	private volatile int writer;  // current writer - will be 0 or 1
	private volatile int waitingWriters;
}
```

readers	Keeps a count of the number of readers reading from a particular Element
writer	Checks if current Element has been locked for a write operation
waitingWriters	Keeps a count of the number of writers waiting to lock an Element

lockRead:
```
while(waitingWriters > 0 || writer > 0)
{ wait(); }
readers++;
```

lockWrite:
```
waitingWriters++;
while(readers > 0 || writer > 0)
{ wait(); }
writer++; waitingWriters--;
```

unlockRead:
```
readers--;
notifyAll();
```

unlockWrite:
```
writer--;
notifyAll();
```

In ***lockRead()*** method, readers have to wait to acquire the lock if another writer is waiting to acquire it. In ***lockWrite()*** method, a writer needs to wait for all current readers to release the lock before it can acquire it. Thus a writer is not starved for lock acquisition, preservation a fairness notion.