**ASSIGNMENT - 1**

**Introduction:**

Welcome to the first programming assignment of Fundamentals of Programming Languages.

This homework is divided into 4 parts, each of which has an associated point total. The total of all parts combines to 100 points. In this homework you will be exploring threading and locking in Java.

The homework is due on **Wednesday September 18th at 11:59 PM.**

Part-1: The Building the Data Structure (15 Points)

Part-2: Concurrency Control

Part 2.0: Design Document (10 Points)

Part 2.1: Coarse-grained Locking (25 Points)  
Part 2.2: Fine-grained Locking (25 Points)  
Part 2.3: Read/Write Locks (25 Points)

Note: Please adhere to the API that we request you to follow as you will be graded exactly based on them. If you do not follow the API you will be assigned 0 points.

**Requirements and Submission:**

All files should have your name and UB email address in comments at the top of the file prior to any code you have written. Create one tarball which should be named with your UBIT id, this tarball should contain all your source files and your design document.

Submit using **submit\_cse505 <filename>**

For details on how to submit, refer to:

<https://wiki.cse.buffalo.edu/services/content/submit-script>

<https://wiki.cse.buffalo.edu/services/content/student-servers>

**Part 1 - Building the Data Structures: 15 Points**

For this part you will build a basic Circular Doubly Linked (CDL) List in Java with the following specification.

**public** **class** CDLList<T> {

**public** CDLList(T v) {}

**public** **class** Element {

**public** T value() {}

}

**public** Element head() {}

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

**public** **class** Cursor {

**public** Element current() {}

**public** **void** previous() {}

**public** **void** next() {}

**public** Writer writer() {}

}

**public** **class** Writer {

**public** **boolean** insertBefore(T val) {}

**public** **boolean** insertAfter(T val) {}

}

}

**Part 2 - Concurrency Control:**

2.0 Design Document **10 Points**

You must describe the algorithms you will use for implementing Fine-grained locking (2.2) and the RW Lock (2.3). Specifically, for the Fine-grained Locking you should list what data each lock protects, how you handle deadlocks, and how you prevent race conditions. For the RW Lock you should list how your Lock implements the desired semantics. The design document should be about a page but no longer than 2 pages.

2.1 Coarse-grained Locking **25 Points**

Implement a subclass of CDLList, CDLCoarse, which supports coarse-grained locking (one lock per list) of your data structure. This means you will create only one lock regardless of how many elements are in the list. This lock should be used to make sure there are no data races and no chance of deadlock in your structure.

2.2 Fine-grained Locking **25 Points**

Implement a subclass of CDLList, CDLListFine, which supports fine-grained locking of your data structure.

* 2.2.1 **15 Points**  
  Fine-grained locking with the following assumptions:
  + *if one of the required locks is acquired, then all the required locks will be acquired*
  + *once locked, an element cannot be changed or deleted*
* 2.2.2 **10 Points**  
  Fine-grained locking which incorporates release of acquired locks (when above assumptions fail)

**An example of use of 2.1 and 2.2 is as follows:**

CDLList<String> f; CDLList<String>.Cursor c;

f = **new** CDLList<String>(“hi”);

c = f.reader( f.head() );

c.next();

f.head() == c.curr(); // a circular list

c.writer().insertBefore(“bye”);

**public** **class** CDLList<T> {

// Create a CDLList with one value

**public** CDLList(T v) {…}

// If deleted an Element becomes invalid. Any access to an invalid element

// throws an exception

**public** **class** Element {

// return the element’s value

**public** T value() {…}

}

// Return the head of the list. Never null

**public** Element head() {…}

// Return a cursor at element from in the list

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

// A cursor on an invalid element throws an exception.

**public** **class** Cursor {

// Return the current element.

**public** Element curr() {…}

// Move to the previous element.

**public** **void** prev() {…}

// Move to the next element

**public** **void** next() {…}

// Returns a writer at the current element

**public** Writer writer() {…}

}

// A cursor on an invalid element throws an exception.

**public** **class** Writer {

// Add before the current element.

**public** **boolean** insertBefore(T val) {…}

// Add after the current element.

**public** **boolean** insertAfter(T val) {…}

}

}

2.3 Creating a Read/Write Lock **25 Points**

Implement a version of the CDLList that is protected by your own R/W locks in place of fine grain locks and the coarse grain lock. Like 2.1 and 2.2, you still need to implement two additional classes that are subclasses of CDLList, CDLListFineRW and CDLCoarseRW.

(1) Note, you must first implement a R/W lock which adheres to the following semantics:

(a) In the implementation of RW lock, you must define four methods: lockRead, unlockRead, lockWrite, unlockWrite.

(b) Any number of readers may hold a R/W lock, but only one writer may hold the lock.

(c) Readers and writers may not hold the lock at the same time.

(d) The locks must preserve a notion of “fairness.”

10 points will be given for parts (a)-(c) and 15 points for part (d) for a totally of 25 points.

(2) We will define “fairness” as follows: when many readers hold a lock and a writer attempts to acquire it, no further readers will acquire the lock until that writer is allowed to proceed. The writer, however, still must wait until the current readers, which hold the lock are done and release the lock. Once the lock is released writers and readers have equal chance to acquire the lock. This means you do not need to guarantee that a writer acquires the locks, just that it has a chance to acquire the lock.

(3) For this assignment, assume that Cursor objects will never be shared between different threads. This means you do not need to build in concurrency control into the cursors.

(4) The locks do not need to be re-entrant and you may read from an object if you have acquired its write lock.

(5) You should utilize the **Java monitors** to guard the internal structure of the lock and leverage **wait** and **notify** to handle blocking threads.

**Extra Credit**  **10 Points**

Extra credit earned in this assignment can be applied to points you have missed either in this assignment or in future assignments. Note that extra credit points can only be applied to homework. 10 bonus points for successfully implementing delete function in your data structure. Deleted elements must be carefully handled in all functions (Part 1, Part 2).

Please note that there will be **no partial credit** for this section, you either get all of the points or none.

**public** **class** Writer {

// Delete the current element and becomes invalid.

**public** **boolean** delete() {…}

// Add before the current element.

**public** **boolean** insertBefore(T val) {…}

// Add after the current element.

**public** **boolean** insertAfter(T val) {…}

}