**README**

**There are three kinds of operations on singly-linked list: search, insert and remove. Searching threads merely examine the list; hence they can execute concurrently with each other. Inserting threads add new items to the front of the list; insertions must be mutually exclusive to preclude two inserters from inserting new items at about the same time. However, one insert can proceed in parallel with any number of searches. Finally, the remove operation removes items from anywhere in the list. At most one thread can remove items at a time, and a remove must also be mutually exclusive with searches and inserts**

**1: Variables and their interpretation with initial value**

Variables and their representation

Variable that represents search operation on linked list : ns

Variable that represents insert operation on linked list : ni

Variable that represents remove operation on linked list: nr

Lock operation variable: lock

Insert Condition variable : insertCond

Remove Condition variable: removeCond

Search Condition variable: searchCond

Initial values

**private** **int** ni = 0;

**private** **int** nr = 0;

**private** **int** ns = 0;

**private** **final** ReentrantLock lock;

**private** **final** Condition insertCond;

**private** **final** Condition removeCond;

**private** **final** Condition searchCond;

**2 Invariant that captures the synchronization requirements above**

Invariant as logical formula

ni ≥0 ∧ nr ≥ 0 ∧ ns≥0 ∧ ((ni==1 ∧ nr == 0 )∨(nr==1 ∧ ni ==0 ∧ ns ==0) ∨ (nr==0))

Invariant in English words

* insert, remove and search should always be non negative.
* At any given point of time if an insert is present then only remove should be absent.
* At any given point of time if a remove is present then insert and search should be absent.
* At any given point of time remove should be absent if and only if a search is present.

**3 Implementations for start\_search, end\_search, start\_insert, end\_insert, start\_remove, and end\_remove in terms of atomic\_await statements**

void start\_search (){<await (r==0) s++>}

void end\_search() {<await(s>0) s-->}

void start\_insert() { <await(i==0 && r ==0) i++ > }

void end\_insert() { <await(i>0) i -- >}

void start\_remove() { < await (r==0 && i == 0 && s==0) r++ >}

void end\_remove(){< await(r>0) r-- >}

**General information**

There will be conflict operations because of the first pointer. First pointer is being used by insert and search both. Since search and insert can run simultaneously insert writes to first pointer and search reads the first pointer in the for loop when it initialises current to first and changes current to first->next thus, there will be data race created by these conflict operations since search and insert can run in any order . Changing first pointer to volatile will solve these data races. If there is an insert while search is happening search will have the consistent old value of first since we have changed it to volatile.

No the new inserts will not be safely published since first is neither static nor volatile. So in order to publish it safely we can change first to volatile.

This class can be used in real practice but it will depend upon how frequent write are occurring as he locking used will bottle neck the performance. If there are frequent writes then following con's are there

Cons :

Locking the whole linked list will slow down insert and search operations.

Handover locking will better perform better than locking the whole linked list as we have done in the current implementation. If we use optimistic locking when no of inserts are less it will improve the performance.

Pros :

Since inserts and removes are mutually exclusive therefore there will be no data race present.

Object will be safely published, search can occur in parallel to inserts so it speeds up the process.