Assignment 3

Objectives

In this assignment, we will use locks to implement a thread-safe linked list.

Background

A linked list is data structure that organizes a set of elements, called *nodes*, in a linear sequence. Linked lists do not keep the sequence of nodes in a contiguous location in physical memory as arrays do. Instead, each node is a separate entity that also contains a pointer to the next node in the list. Each node has two fields: a data field which contains the data, and a link or reference field, which links it to the other elements in the sequence. The simplest example of a linked list is a singly linked list. Each node in a singly linked list keeps a reference to the next immediate node, as seen in Figure 1. The list shown contains 3 nodes. The first node (data = 12) links to the second (data = 99), the second to the third (data = 37), and the last node (data = 37) links to null. Linked lists are useful because the computation complexity of addition and removal of an element (without indexing) is O(1) as opposed to arrays that require O(N) operations. They also have low memory footprint, as nodes are allocated at runtime.

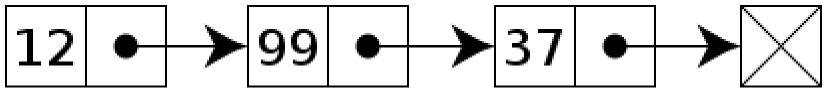

Figure 1: An example of a singly linked list

Assignment

You will design a thread-safe singly linked list. A thread-safe linked list allows multiple threads to operate on the linked list in parallel without corrupting the list. You will design the `insert`, `delete`, and `search` operations of the list, starting from the single threaded baseline given below. Assume the list is already initialized and all the elements (if any) are sorted. The variable `head` is a special `node_t*` whose `next` points to the first element in the list and its `val` is set to `INT_MIN`. If `head->next` is `NULL`, it means the list is empty. `head` is not deleted and it's always there even if the list is empty.

This assignment has no implementation component, you will only hand in a report.

```c
// Linked list struct
typedef struct node {
    int val;
    struct node *next;
    int to_remove;
} node_t;

/* This function inserts a new given element at the right position of a given linked list.
 * It returns 0 on a successful insertion, and -1 if the list already has that value.
 */
int insert(node_t *head, int val) {
    node_t *previous, *current;
    current = head;

    while (current && current->val < val) {
        previous = current;
        current  = current->next;
    }

    if (current && current->val == val) { // This value already exists!
        return -1;
    }

    // Here is the right position to insert the new node.
    node_t *new_node;
    new_node = malloc(sizeof(node_t));
    new_node->val = val;
    new_node->next = current;
    new_node->to_remove = 0

    previous->next = new_node;
    return 0;
}

/* This function removes the specified element of a given linked list.
 * The value of that element is returned if the element is found; otherwise it returns -1.
 */
int delete(node_t *head, int val) {
    node_t *previous, *current;

    if (head->next == NULL) { // The list is empty.
        return -1;
    }

    previous = head;
    current = head->next;
    while (current) {
        if (current->val == val) {
            previous->next = current->next;
            current->to_remove = 1; // Another system component will free this node later
            return val;
        }

        previous = current;
        current  = current->next;
    }
    return -1;
}
```

```c
/* This function searches for a specified element in a given linked list.
 * It returns zero if the element is found; otherwise it returns -1.
 */
int search(node_t *head, int val) {
    node_t *current = head->next;

    while (current) {
        if (current->val == val) {
            return 0;
        }
        current  = current->next;
    }

    return -1;
}
```

Report

The report will describe the structure of your solution. It should answer the following questions:

1. Identify all data races that occur in the `insert`, `delete`, and `search` baseline functions when they are executed concurrently. Please note there might be races between two threads executing the same operations or races between threads executing different operations, for instance one executing insert and the other executing delete. You can describe the races in your own words or using "happens-before analysis". [10 points]

2. The simplest solution to make the operations thread-safe is to add a single lock that controls access to the entire data structure. Modify the code to have this lock. [10 points]

3. What is the biggest performance bottleneck of the approach taken in step 2? [10 points]

4. In this step, create a thread-safe list by adding a separate lock to each list element. Modify the code to use these locks during the list traversal. Describe your design, showing how each data race is dealt with. Make sure your solution is deadlock free. [20 points]

5. Compare the performance of your designed solutions in step 2 and step 4. How did the new design resolve the problem we had with the previous design? What is the main performance bottleneck of the new design? [15 points]

6. Now, design a thread-safe list that performs the list traversal without locking elements. To maintain correctness your implementation might do some extra steps. Modify the code to only lock elements after the desired element is found. Describe your design, showing how each data race is dealt with. Make sure your solution is deadlock free. [20 points]

7. How does the design in step 6 solve the performance issues identified in question 5? Is there any situation in which the question 4's design is better than question 6's. [15 points]

Deliverable

You only need to submit one PDF file containing your answered to the above questions called `a3_<yourGroupID>.pdf` **(this is the only accepted format!)** and upload it on Moodle. Please include your names in the file.