Lucien Iseli Michaël, 274999 & Loris Pilotto, 262651

Assignement 3 - CS-307

1.

	Insert	**Delete**	**Search**
Insert	Data Race	Data Race	Data Race
Delete	Data Race	Data Race	Data Race
Search	Data Race	Data Race	No Data Race

- **Insert-insert** : There is a data race, for example if the list is HEAD -> 1 -> X, and thread T0 and T1 want to insert 2 and 3 respectively, they will both write to 1's `next` to point it to the element they are adding -> data race.
- **Delete-delete** : There clearly is a data race, if two threads want to delete the same element they will both write to the previous element's `next` which will be the same for the two threads -> data race
- **Search-search** : Search doesn't write to anything non-local -> no data race.
- **Insert-delete and delete-insert** : There is a data race, for example if the list is HEAD -> 1 -> 3 -> X, and thread T0 wants to insert 2 and thread T1 wants to remove 3. They both will write to their previous

element's next which will be 1 for T0 because he wants to add 2 right after 1, and 1 for T1 because it's the element before 1. -> They will write to the same element's next -> data race. This is symmetric so it's the same for delete-insert.

- **Insert-search and search-insert** : There is a data race, for example if the list is HEAD -> 1 -> 4 -> X and thread T0 wants to insert 2 and T1 wants to search 4. T0 wants to write to 1's next and T1 wants to read 1's next to get to the next element to continue searching. -> There is a data race as one thread reads to a memory location another thread writes to. This is symmetric so it's the same for search-insert.

- **Delete-search and search-delete** : There is a data race, for example if the list is HEAD -> 1 -> 4 -> X and thread T0 wants to delete 4 and T1 wants to search 4. T0 wants to write to 1's next and T1 wants to read 1's next to get tot he next element to continue searching. -> There is a data race as one thread reads to a memory location another thread writes to. This is symmetric so it's the same for search-delete.

2.

```
#include <omp.h>

omp_lock_t lock;
omp_init_lock(&lock);

// 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) {
  omp_set_lock(&lock);
  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!
    omp_unset_lock(&lock);
    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;

  omp_unset_lock(&lock);
  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) {
  omp_set_lock( & lock);
  node_t* previous, *current;

  if (head->next == NULL) { // The list is empty.
    omp_unset_lock(&lock);
    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
      omp_unset_lock(&lock);
      return val;
    }
```

```
    previous = current;
    current = current-> next;

  }

  omp_unset_lock(&lock);
  return -1;
}

/* 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) {
  omp_set_lock(&lock);
  node_t* current = head->next;

  while (current) {
    if (current-> val == val) {
      omp_unset_lock(&lock);
      return 0;
    }

    current = current->next;
  }

  omp_unset_lock(&lock);
  return -1;
}
```

3.

The biggest performance bottleneck of the approach taken in step 2 is the fact that we can’t execute several functions at the same time since they all share the same lock. It is not the most efficient way to achieve thread-safety since not all combinations of them have a data race. Here, we cannot do two search at the same time even though they have no data race.

4.

```
#include <omp.h>

// Linked list struct
typedef struct node {
    int val;
    struct node* next;
    int to_remove;
    omp_lock_t* lock;
} node_t;

omp_init_lock(head->lock); //We need to
initialize head’s lock

/* 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 = null;
    node_t* previousTmp = null;
    node_t* current;
```

```
    current = head;
    omp_set_lock(current->lock);

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

        omp_lock_t(current->lock) ;
        if(previousTmp) {
            omp_unset_lock(previousTmp->lock);
        }
    }

    if (current && current->val == val) { // This
value already exists!
        if(previous) {
            omp_unset_lock(previous->lock);
        }
        omp_unset_lock(current->lock);
        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;
    omp_init_lock(new_node->lock);
    previous->next = new_node;
```

```
    if(current) {
        omp_unlock_t(current->lock);
    }
    omp_unlock_t(previous->lock);
    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, *previousTmp;
    omp_set_lock(head->lock);

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

    previous = head;
    current = head->next;

    if(current) {
        omp_set_lock(current->lock);
    }

    while (current) {
        if (current->val == val) {
            previous->next = current->next;
            current->to_remove = 1; // Another
```

```
system component will free this node later

            omp_unset_lock(current->lock);
            omp_destroy_lock(current->lock);
            omp_unset_lock(previous->lock);

            return val;
        }
        previousTmp = previous;
        previous = current;
        current  = current->next;
        omp_set_lock(current->lock);

        if(previousTmp) {
            omp_unset_lock(previousTmp->lock) ;
        }
    }
    return -1;
}

/* 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;
    node_t *previous = null;

    if(current) {
        omp_set_lock(current->lock);
    }
```

```
    while (current) {
        if (current->val == val) {
            omp_unset_lock(current->lock);
            return 0;
        }
        previous = current;
        current  = current->next;
        if(current) {
            omp_set_lock(current->lock);
        }
        omp_unset_lock(previous->lock);
    }
    return -1;
}
```

We designed our thread-safe list as explained in the course. For the insert and delete functions we always lock the previous and current node while going through the linked list. This method is called « hand over hand » locking and ensure the list to be thread-safe. We have to be carefull to unset the locks once we move forward in the linked list and before leaving each functions.

For the search function we only locked the current node because we don’t need to have access to the previous node so it’s useless to use the «hand over hand» method here. Having locked the current value is enough.

5.

The biggest performance bottleneck of the approach taken in step two was the fact we couldn’t execute several functions at the same time since they all share the same lock. Now each locks are related with a single node of the linked list so several functions (i.e. search, delete and insert) can access the list now. If we execute several search/delete/insert on different threads the performance of the new design will be increased a lot, especially if they don’t need to ‘cross’ themselves. i.e. the first method to execute has to go at the end of the list, the next one a bit before, and so on. This way they don’t have to wait for the previous method to finish before being able to continue traversing the list.

However, now one function takes more time to execute, because software locks take some time and with this design we lock/unlock locks in O(n), where n is the size of the list, whereas before it was O(1). So the methods are now slower alone, but may execute faster if we have multiple ones in parallel.

Another performance bottleneck is if the methods are in the worst order possible, i.e. the method that executes make all the next ones wait, because they need to traverse the list and the first one is locking it.

6.

```
#include <omp.h>

// Linked list struct
typedef struct node {
    int val;
```

```
    struct node* next;
    int to_remove;
    omp_lock_t* lock ;
} node_t;

omp_init_lock(head->lock); //We need to
initialize head’s lock

/* This function checks if the list is valid with
respect to curr and prev
 * i.e. that the previous is still accessible and
that its next is curr
 */
int validate(node_t* head, node_t* prev, node_t*
curr) {
    node_t* node = head;

    while(node && node->val <= prev->val) {
        if(node == prev) {
            return node->next == curr;
        }
        node = node->next;
    }
    return 0;
}

/* 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;
    }

    omp_set_lock(previous->lock);
    if(current) {
        omp_set_lock(current->lock);
    }

    if(!validate(head, previous, current)) {
        omp_unset_lock(previous->lock);
        if(current) {
            omp_unset_lock(current->lock);
        }

        //We restart because the state of the
list is not as expected
        return insert(head, val);
    }

    if (current && current->val == val) { // This
value already exists!
        omp_unset_lock(previous->lock);
        omp_unset_lock(current->lock);
        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;
    omp_init_lock(new_node->lock);

    previous->next = new_node;
    omp_unset_lock(previous->lock);
    if(current) {
        omp_unset_lock(current->lock);
    }

    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) {
```

```
            omp_set_lock(previous->lock);
            omp_set_lock(current->lock);

            if(!validate(head, previous,
current)) {
                omp_unset_lock(previous->lock);
                omp_unset_lock(current->lock);

                //We restart because the state of
the list is not as expected
                return delete(head, val);
            }

            previous->next = current->next;
            current->to_remove = 1; // Another
system component will free this node later

            omp_unset_lock(previous->lock);
            omp_unset_lock(current->lock);
            omp_destroy_lock(current->lock);

            return val;
        }

        previous = current;
        current  = current->next;
    }

    return -1;
}

/* 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) {
            omp_set_lock(current->lock);

            if(!validate(head, current, current-
>next)) {
                omp_unset_lock(current->lock);

                //We restart because the state of
the list is not as expected
                return search(head, val);
            }

            omp_unset_lock(current->lock);
            return 0;
        }

        current  = current->next;
    }

    return -1;
}
```

In the new implementation, delete and insert first search for the node we want, then it locks this node and the previous one. Then the function call “validate” to check that the previous node is still accessible and that its next

node is indeed the node we were looking for. We loop until those conditions are satisfied. If “validate” is true it means that the two locks are correctly set and the list is in the state we expect it to be in. We can then delete/insert the node safely and then free the locks.

The function search work as delete/insert but we only need to lock the node we are looking for since we don’t need to modify the previous node once we found the node we were looking for.

This solution should be deadlock free, because we look for nodes -> only reads and when we find the node we want we lock it and the previous one. So no one can modify them. Then we check that the list is in a state that makes sense, i.e. that the nodes we have still are in the list. If they still are in the list and linked then we can safely apply changes, if they don’t we try again. Since the nodes are locked, and they are still in a valid state in the list, their state cannot be changed until we unlock them. So we can safely modify them.

7.

Now that we don’t do lock during the traversal of the list, the traversal goes faster and has less waiting time. So the traversal of the list should be faster, however now we have to traverse it twice to do anything, once to find the node and a second time to validate the state. So if there are a lot of contention this implementation will probably go faster as it needs way less locking/unlocking interaction. However if

there is no contention the solution in step 4 is probably faster, as acquiring the lock will be very fast and we will only traverse the list once.