SAINT PETERSBURG ELECTROTECHNICAL UNIVERSITY

"Computer Science and Knowledge Discovery"

2nd Semester Study

Group - 6300

Homework 3:

Subject:	Parallel Computing
Topic:	Lock free data structures and algorithms
Submitted To:	Prof. **Natalya V. Shevskaya**
Submitted by:	Shahrukh Sultan

Lock-Free Data Structures & Algorithms:

Blocking and Non-blocking Concurrent Programming:

In parallel computing programs when different threads need to access the same data, we must ensure that the data is stored in a coherent way. We can achieve this by locking i.e. when a thread will acquire a mutex to read/write the data other threads can't access the data at the same time as they are blocked and waiting for the mutex to be released. Algorithms and data structures that use mutexes and this blocking condition is called blocking data structures and algorithms. Data structures and algorithms that don't use blocking library functions are of non-blocking type.

Lock free data structure:

Lock free programming is the implementation of algorithms and data structures that do not acquire locks or mutexes. It's a kind of non-blocking concurrent programming approach.

In implementation of lock free data structures, more than one thread must be able access the data concurrently. Different threads have to be able to perform different tasks. A lock free queue might allow one thread to push and one to pop but break if two threads try to push new items at the same time. If one threads accessing the data is suspended by the scheduler midway through its operation, the other thread must still be able to complete their operations without waiting for the suspended thread.

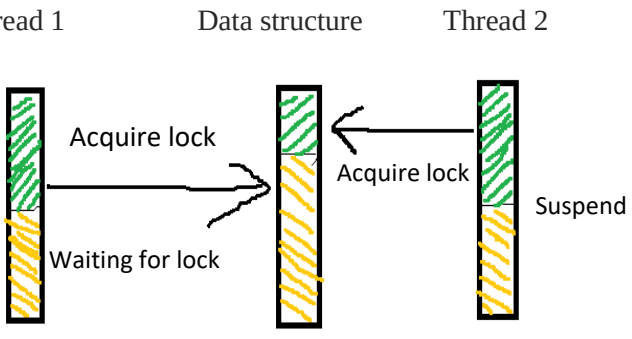


Fig: Locking strategy

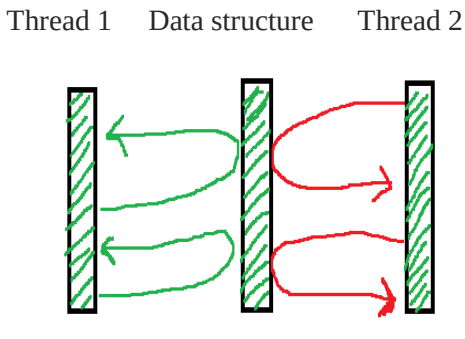


Fig: Lock free strategy

Implementation of Lock-Free Data Structures & Algorithms:

Implementing Lock free stacks in C++:

Let's design a concurrent stack data structure where a thread will create a new version of the top of the stack and if no other thread has modified the stack, the change will be made public. To accomplish this, a compare and swap (CAS) instruction is required to do the comparison and the write operations automatically.

This can be done by following steps

1. Create a new node.
2. Set its next pointer to the current top node.
3. Set the top node to point to it.

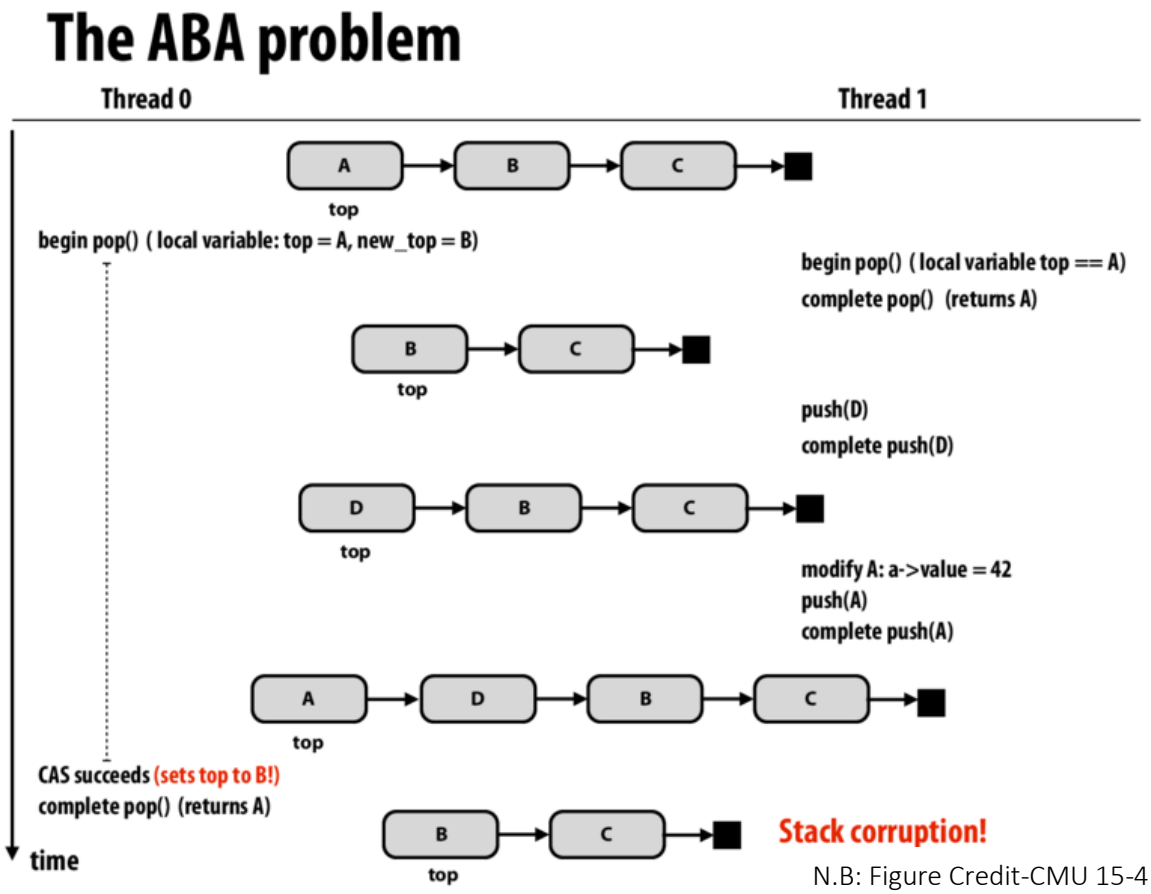
To check that the top of the stack is not changed in between creating the new top and writing it to the memory, this is the pseudocode in C++ for Push operations.

Push

```
void push (stack *s, Node *n){
        while (1) {
            Node *old_top =s->top;
            n->next = old_top;
            if(compare and swap (&s->top, old_top, n) == old_top)
                return:
         }
  }
```

But CAS instruction doesn't necessarily succeed because another thread may perform its own CAS and change **s->top**. If this happens the CAS will see that **old_top** has changed and will try to perform the insert again with the new value.

This issue can be visualized as ABA problem



N.B: Figure Credit-CMU 15-418, spring 2013

In steps the problem can be exposed as

- Thread 0 begins a pop and sees "A" as the top, followed by "B".
- Thread 1 begins and completes a pop, returning "A".
- Thread 1 begins and completes a push of "D".
- Thread 1 pushes "A" back onto the stack and completes.
- Thread 0 sees that "A" is on top and returns "A", setting the new top to "B".
- Node D is lost.

This issue can be resolved by repeated push pop operations and using a counter keep track of the number of pops.

Pop

```
Node *pop(Stack *s) {
  while (1) {
    Node *top = s->top
    int pop_count = s->pop_count;
    if (top == NULL)
      return NULL;
    Node *new_top = top->next;
    if (double_compare_and_swap(&s->top, top, new_top, &s->pop_count,
        pop_count, pop_count + 1));
      return top;
  }
}
```

Here, a double-CAS is used to update both the stack and the counter, only if both are unchanged.

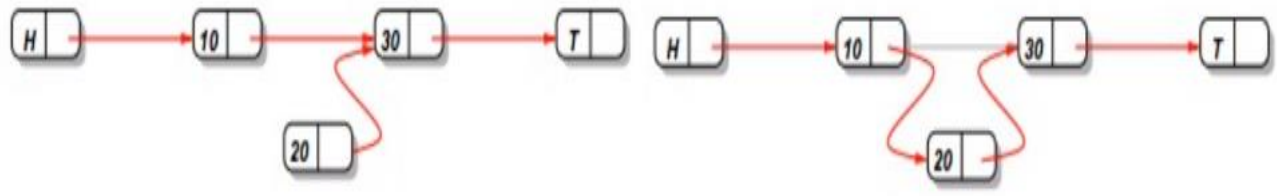
Implementing a lock-free linked list:

To implement a lock free linked list data structure first for insertion we need to find the correct position in the list. Assume we have found this location and we are inserting Node *n after Node *p. The insertion code is

Insertion:

```
while (1) {
  n->next = p->next;
  Node *old_next = p->next;
  if (compare_and_swap(&p->next, old_next, n) == old_next)
    return;
}
```

Here we need compare_and_swap to update the next pointer for the node we're inserting after (*p) to ensure that another thread hasn't modified the same position. This process can be visualized as shown below.



Deletion:

After insertion for deletion there is an issue that simultaneous insertion and deletion is not easy to account for. If we have "A"->"B"->"C", a problem case would be

- Thread 0 begins to delete "B" from the list after "A".
- Thread 1 begins to insert "D" after "B".
- Thread 1 points "B" to "D".
- Thread 0 points "A" to "C"
- The delete operation removed "B" and "D".

To fix this, there must be a way to ensure that "A" is also unchanged when inserting after "B" and how to react when this happens.