ETU "LETI"

Saint Petersburg Electrotechnical University "ETU LETI" Russian Federation, Saint Petersburg May, 2021

Lock-free data structures and algorithms

(Parallel computing)

Submitted by

Ryleev Vladislav Valerevich

Chubchik Alexander Olegovich

Master Group: 6300

Submitted to

Shevskaya Natalya Vladimirovna

Main (general) definitions for terms which are important to know for Lock-free data structures and algorithms.

Non-blocking synchronization is an approach to parallel programming on symmetric multiprocessor systems that departs from traditional locking primitives such as semaphores, mutexes, and events. Access sharing between threads is done through atomic operations and special, designed for a specific task, locking mechanisms.

The Lock-Free property guarantees that at least some thread is doing progress on its work. In theory this means that a method may take an infinite amount of operations to complete, but in practice it takes a short amount, otherwise it won't be much useful.
Definition: A method is Lock-Free if it guarantees that infinitely often some thread calling this method finishes in a finite number of steps.

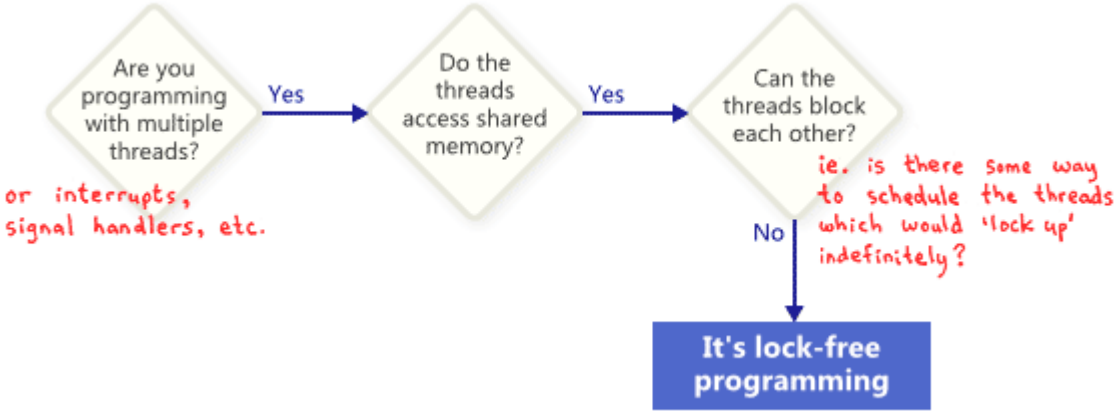
Symmetric multiprocessing (**SMP**) involves a multiprocessor computer hardware and software architecture where two or more identical processors are connected to a single, shared main memory, have full access to all input and output devices, and are controlled by a single operating system instance that treats all processors equally, reserving none for special purposes. Most multiprocessor systems today use an SMP architecture. In the case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors.

Atomic operation is an operation that will always be executed without any other process being able to read or change state that is read or changed during the operation. It is effectively executed as a single step, and is an important quality in a number of algorithms that deal with multiple independent processes, both in synchronization and algorithms that update shared data without requiring synchronization.

Explanation for main idea and problem

Let’s describe lock-free programming as programming without mutexes, which are also referred to as locks. That’s true, but it’s only part of the story. The generally accepted definition, based on academic literature, is a bit more broad. At its essence, lock-free is a property used to describe some code, without saying too much about how that code was actually written.

Basically, if some part of your program satisfies the following conditions, then that part can rightfully be considered lock-free. Conversely, if a given part of your code doesn’t satisfy these conditions, then that part is not lock-free.



In this sense, the lock in lock-free does not refer directly to mutexes, but rather to the possibility of "locking up" the entire application in some way, whether it's deadlock, livelock – or even due to hypothetical thread scheduling decisions made by your worst enemy. That last point sounds funny, but it's key. Shared mutexes are ruled out trivially, because as soon as one thread obtains the mutex, your worst enemy could simply never schedule that thread again. Of course, real operating systems don't work that way – we're merely defining terms.

The promise of a lock-free algorithm seems remarkable in theory. Concurrent threads can modify the same object, and even if some thread or set of threads stalls or stops completely in the middle of an operation, the remaining threads will carry on as if nothing were the matter. Any interleaving of operations still guarantees of forward progress. It seems like the holy grail of multi-threaded programming.

This stands in contrast to the traditional method of placing locks around "critical sections" of code. Locking prevents multiple threads from entering these critical sections of code at the same time. For highly concurrent applications, locking can constitute a serious bottleneck. Lock-free programming aims to solve concurrency problems without locks. Instead, lock-free algorithms rely on atomic primitives such as the classic "compare-and-swap" which atomically performs the following:

```
bool CompareAndSwap(Value* addr, Value oldVal, Value newVal){
if(*addr == oldVal){
*addr = newVal;
return true;
}else{
return false;
}
}
```

The biggest drawbacks to using lock-free approaches are:

- Lock-free algorithms are not always practical.
- Writing lock-free code is difficult.
- Writing correct lock-free code is extremely difficult.

To illustrate the third point above, let's break down a typical first attempt at writing a lock-free stack. The idea is to use a linked-list to store nodes, and use CompareAndSwap to modify the head of the list, in order to prevent multiple threads from interfering with each other.

To push an element, we first create a new node to hold our data. We point this node at the current head of the stack, then use CompareAndSwap to point the head of the stack to the new element. The CompareAndSwap operation ensures that we only change the head of the stack if our new node correctly points to the old head (since an interleaving thread could have changed it).

To pop an element, we snapshot the current head, then replace the head with the head's next node. We again use CompareAndSwap to make sure we only change the head if its value is equal to our snapshot.

The code, in C++

```
template
class LockFreeStack{
struct Node{
Entry* entry;
Node* next;
};

Node* m_head;

void Push(Entry* e){
```

```
Node* n = new Node;
n->entry = e;
do{
n->next = m_head;
}while(!CompareAndSwap(&m_head, n->next, n));
}

Entry* Pop(){
Node* old_head;
Entry* result;
do{
old_head = m_head;
if(old_head == NULL){
return NULL;
}
}while(!CompareAndSwap(&m_head, old_head, old_head->next));

result = old_head->entry;
delete old_head;
return result;
}
}
```

Unfortunately, this stack is riddled with errors.

Segfault

The Push() method allocates memory to store linked-list information, and the Pop() method deallocates the memory. However, between the time a thread obtains a pointer to a node on line 22, and subsequently accesses the node on line 26, another thread could have removed and deleted the node, and the thread will crash. A safe way to reclaim the memory is needed.

Corruption

The CompareAndSwap method makes no guarantees about whether the value has changed, only that the new value is the same as the old value. If the value is snapshotted on line 22, then changes, then changes back to the original value, the CompareAndSwap will succeed. This is known as the ABA problem. Suppose the top two nodes on the stack are **A** and **C**. Consider the following sequence of operations:

- Thread 1 calls pop, and on line 22 reads m_head (**A**), on line 26 reads old_head->next (**C**), and then blocks just before calling CompareAndSwap.
- Thread 2 calls pop, and removes node **A**.
- Thread 2 calls push, and pushes a new node **B**.
- Thread 2 calls push again, and this time pushes a new node occupying the reclaimed memory from **A**.
- Thread 1 wakes up, and calls CompareAndSwap.

The CompareAndSwap on line 26 succeeds even though m_head has changed 3 times, because all it checks is that old_head is equal to m_head. This is bad because now m_head will point to **C**, even though **B** should be the new head.

Examples in pseudocode about how we can implement it in practice

```
ENQUEUE(x)
   q ← new record
   q^.value ← x
   q^.next ← NULL
   repeat
      p ← tail
      succ ← COMPARE&SWAP(p^.next, NULL, q)
      if succ ≠ TRUE
         COMPARE&SWAP(tail, p, p^.next)
   until succ = TRUE
   COMPARE&SWAP(tail,p,q)
end

DEQUEUE()
   repeat
      p ← head
      if p^.next = NULL
         error queue empty
   until COMPARE&SWAP(head, p, p^.next)
   return p^.next^.value
end
```

For implementation:

Some non-blocking data structures are weak enough to be implemented without special atomic primitives. These include:

- A single-reader single-writer ring buffer FIFO, with a size which evenly divides the overflow of one of the available unsigned integer types, can unconditionally be implemented safely using only a memory barrier.
- Read-copy-update with a single writer and any number of readers. (The readers are wait-free; the writer is usually lock-free, until it needs to reclaim memory).
- Read-copy-update with multiple writers and any number of readers. (The readers are wait-free; multiple writers generally serialize with a lock and are not obstruction-free).