**Program**

**Sought**

Use the Tervel framework to implement a linked list ﬁrst-in-ﬁrst-out queue container based on the queue described in chapter 10.5 of “The Art of Multiprocessor Programing” book. This implementation should use Tervel’s memory management framework to safely return allocated objects to system allocator. After implementing the lock-free version, use Tervel’s progress assurance scheme to make a wait-free version. A partial implementation is provided for your convenience.

**Solution**

The queue data structure is one of the basic data structures used in numerous program design. Thus, an effective non-blocking queue implementation based on compare-and-swap (CAS) operation could serve the basis for many data structures. In this assignment, I implemented a C++ version of non-blocking queue using the solution presented in the book. The algorithm implements the queue as a singly-linked list with Head and Tail pointers. The Head always points to a dummy node, which is the first node in the list and Tail points to the last node in the list. The algorithm uses compare and swap to avoid the ABA problem.

**Part 1: Lock-Free Implementation**

In order to implement the Lock-free linked list in Tervel framework, a part of code was implemented by TA. I referred to book and also [1] to completely understand how the lock-free linked list works. To implement the enq() method, I modified bool Queue<T>::enqueue(T &value) function as below:

template<typename T>

bool Queue<T>::enqueue(T &value) {

Node \*node = new Node(value);

while (true) {

Accessor access;

Node tailnode\_;

if (access.init(&tailnode\_, &tail\_) == false) {

continue;

}

Node \*last = access.ptr();

Node \*next = access.ptr\_next();

if (last == tail()) {

if (next == nullptr) {

if (last->next\_.compare\_exchange\_strong(next, node)) {

set\_tail(last, node);

return true;

}

} else {

set\_tail(last, next);

}

}

}

}

Initially, a new node is allocated which copies enqueued value into node. I used access.init to provide safety when dereferenceing a value loaded from tail. Then, the last node is located in the queue. If tail\_.load()->next is not null the set\_tail is used to fix the state. To verify that node is indeed last, I check whether that node has a successor. If so, the thread attempts to append the new node by calling set\_tail(last, next). Otherwise, the compare\_exchange\_strong() is called to append the new node and change the queue’s tail ﬁeld from the prior last node to the current last node.

Thebool Queue<T>::dequeue(Accessor &access)has been modified as below:

template<typename T>

bool Queue<T>::dequeue(Accessor &access) {

while (true) {

Node headnode\_;

if(access.init(&headnode\_,&head\_) == false){

continue;

}

Node \*first = access.ptr();

Node \*last = tail();

Node \*next = access.ptr\_next();

if (first == head()) {

if (first == last) {

if (next == nullptr) {

return false;

}

set\_tail(last, next);

} else {

if (set\_head(first, next)) {

return true;

}

}

}

}

}

First, I used access.init to safeguard access to the value loaded from head. Then, I check to see if the linked-list is empty, return false. Otherwise, I used set\_head to remove the first element and move the head pointer to the next element in the list.

In order to protect the memory on ‘node’ loaded from ‘address’, I implemented the below code in bool Queue<T>::Accessor::init(Node \*node, std::atomic<Node \*> \*address):

template<typename T>

bool Queue<T>::Accessor::init(Node \*node, std::atomic<Node \*> \*address) {

assert(node != nullptr);

node = address->load();

bool res = true;

if (node != nullptr) {

res = tervel::util::memory::hp::HazardPointer::watch(kSlot, node,

reinterpret\_cast<std::atomic<void \*> \*>(address), node);

} else

return true;

if (res)

node\_ = node;

Node \*nextelem = node->next\_.load();

typedef tervel::util::memory::hp::HazardPointer::SlotID SlotID;

static const SlotID watch\_pos = SlotID::SHORTUSE;

if (node == address->load()) {

res = tervel::util::memory::hp::HazardPointer::watch(watch\_pos, nextelem,

reinterpret\_cast<std::atomic<void \*> \*>(&node->next\_), nextelem);

}

if (res)

next\_ = nextelem;

return res;

};

The HazardPointer provided in Tervel framework applies also memory protection on `node->next' and have it only succeed if node is still at `address'.

**Correct behavior in all cases:** The presented algorithm is correct because it satisfies the following properties: *1. The linked list is always connected, 2. Nodes are only inserted after the last node in the linked list, 3. Nodes are only deleted from the beginning of the linked list, 4. Head always points to the first node in the linked list, 5. Tail always points to a node in the linked list.*

The linked list is always connected because once a node is inserted, its next pointer is not set to NULL before it is freed, and no node is freed until it is deleted from the beginning of the list. Furthermore, an inserted node is linked only to a node that has a NULL next pointer, and the only such node in the linked list is the last one. Regarding to deleting a node from the linked list, Head always points to the first node in the list, because it only changes its value to the next node atomically (using compare\_exchange\_strong). When this happens the node it used to point to is considered deleted from the list.

**FIFO property:** This queue is extension of the unbounded total queue. A sentinel node alone in the queue will never be deleted, so each enq() call will succeed. Its implementation prevents method calls from starving by having the quicker threads help the slower threads.

**Lock-free progress:** The lock-free algorithm is non-blocking because if there are non-delayed processes attempting to perform operations on the queue, an operation is guaranteed to complete within finite time.

**Part 2: Wait-Free Implementation**

A highly desired property of any concurrent data structure implementation, and queues in particular, is to ensure that a thread completes its operations in a bounded number of steps, regardless of what other threads are doing. This property is known in the literature as wait-free implementation [2][3]. The wait-free algorithm is achieved by assigning each operation a dynamic age-based priority and making threads with younger operations help older operations to complete. In particular, in the assignment it has been asked to use constructs found in <tervel/util/progress\_assurance.h>. In order to scheme to be effective, each operation in the ProgressAssurance class which may indefinitely prevent the progress of some other operation must call the static function offer\_help. This ensures that if a thread is continually failing its operation, then after a finite number of tries all threads will be helping.

As recommended by TA, I looked at stack implementation file to get the idea how I can make the linked list wait-free. Please find below the implementation code for bool Queue<T>::enqueue(T &value):

bool Queue<T>::enqueue(T &value) {

Node \*node = new Node(value);

tervel::util::ProgressAssurance::check\_for\_announcement();

util::ProgressAssurance::Limit progAssur;

while (!progAssur.isDelayed()) {

Accessor access;

Node accessNode;

if (access.init(&accessNode, &tail\_) == false) {

continue;

}

The most parts of this code is an update on lock-free linked list and wait-free stack implementation.

Node \*last = access.ptr();

Node \*next = access.ptr\_next();

if (last == tail()) {

if (next == nullptr) {

if (last->next\_.compare\_exchange\_strong(next, node)) {

set\_tail(last, node);

return true;

}

} else {

set\_tail(last, next);

}

}

}

EnqueueOp \*op = new EnqueueOp(this, value);

tervel::util::ProgressAssurance::make\_announcement(op);

op->safe\_delete();

size(1);

return true;

}

Thebool Queue<T>::dequeue(Accessor &access)has been modified as below to be wait-free:

template<typename T>

bool Queue<T>::dequeue(Accessor &access) {

tervel::util::ProgressAssurance::check\_for\_announcement();

util::ProgressAssurance::Limit progAssur;

while (!progAssur.isDelayed()) {

Node node;

if (access.init(&node, &head\_) == false) {

continue;

}

Node \*first = access.ptr();

Node \*last = tail();

Node \*next = access.ptr\_next();

if (first == head()) {

if (first == last) {

if (next == nullptr) {

return false;

}

set\_tail(last, next);

} else {

if (set\_head(first, next)) {

return true;

}

}

}

}

DequeueOp \*op = new DequeueOp(this);

tervel::util::ProgressAssurance::make\_announcement(op);

bool res = op->result(access);

if (res)

size(-1);

op->safe\_delete();

return res;

};

This code is also a modified version of lock-free linked list and wait-free stack implementation. In order for the linked list to behave in a wait-free manner, the user must choose a memory allocator that can manage memory in a wait-free manner. Furthermore, this memory manager must be able to handle the ABA problem correctly, because this problem is fundamental to all CAS-based systems. To achieve this the Tervel algorithm uses Michael’s ABA-free approach to safe memory-reclamation, called hazard pointers.

The concept of Hazard pointers works when each thread announce the address of the memory it is about to access. In the implemented algorithm each thread performs an atomic read at a position on Tail and Head and if they are modifying by another thread, the thread writes the address of the node to a global array. The thread then checks to ensure that, between reading the node and writing to the global array, the node was not removed from that location. If it was removed, then the thread retries; this retrying is what makes some other algorithms that use hazard pointers lock-free. In the Tervel algorithm,

the atomic bitmark is used and it is expanded to bound the number of times a retry is attempted.

Michael’s hazard pointer implementation is wait-free if a reference can be placed into the watched address list in a wait-free manner. This consists of reading the contents of an address, storing the value read into the global list, re-reading the contents, and comparing the two values to ensure that they are the same. If they are different it must retry until they are the same. The watch function has been implemented in Tervel algorithm which uses a thread-local variable, threadID, and a global array, watchedNodes, to alert other threads of the node a particular thread is using.Watching is done before any read or write operations on the linked list. Each thread has a unique value form 0 to Threads as their threadID, this corresponds to the position on the watchedNodes array where it stores the node that it is about to use. According to aforementioned information, I used the *watch()* function to implement wait-free linked list.

**References:**

[1] Maged M. Michael Michael L. Scott, “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms,” 1996.

[2] Alex Kogan, Erez Petrank, “Wait-Free Queues With Multiple Enqueuers and Dequeuers,” PPoPP’11.

[3] P. Laborde, S. Feldman, D. Dechev, “A Wait-Free Hash Map,” Int J Parallel Prog, 2014.