Thread Safe Queue (FIFO)

This Thread Safe Queue (FIFO) has following specifications:-

1. Multi-Reader Multi-Writer Model - Concurrent access from multiple writer and reader threads
2. Only one thread can be pushing an element at any given time
3. Only one thread can be popping an element at any given time
4. Remove- Removes one element from queue. Threads must wait for elements to be pushed if the queue is empty
5. Add – Adds one element to the queue. Threads must wait for elements to be popped if the queue is full
6. Clear – Clears any elements in the queue which were added but not yet removed
7. Shutdown – Prevents any blocked or future calls of Add or Remove by throwing an exception

The Queue is made safe for n reader and n writer threads by implementing C++11 multithreading support as follows:-

* **Mutual Exclusion**: Only one thread at a time acquires a given lock on critical section

There are two types of locks used in the code:

1. **std:unique\_lock :**

* Manages a *mutex object* with *unique ownership* in both states: *locked* and *unlocked*.
* Used together with std::condition variables.
* Guarantees an unlocked status on destruction (even if not called explicitly) and in case an exception.

1. **Std:lock\_guard** :

* lock\_guard object will lock the mutex on its construction and unlock on destruction.
* **Condition variable**
* A *condition variable* is an object able to block the calling thread until *notified* to resume.
* Uses a [unique lock](http://www.cplusplus.com/unique_lock) (over a [mutex](http://www.cplusplus.com/mutex)) to lock the thread when one of its wait() is called.
* The thread remains blocked until woken up by another thread that calls notify() on the same condition variable object.
* **Wait & Notify**
* The wait(), notify() and notifyAll() methods can only be called from within a critical section.
* The wait() method releases the lock, and then puts the current Thread into a wait state until some other Thread (that then holds the same lock) calls notifyAll().
* notify\_one() will choose one arbitrary Thread that is waiting on the lock in question and force it out of the wait state into the runnable state.
* **Atomic variable**
* Atomic variable \_shutdown is used to notify threads about queue shutdown.
* If one thread writes to an atomic variable while another thread reads from it, the behavior is well-defined. No race condition.

**Thread Safe Queue Class**

1. #include <mutex>
2. #include <condition\_variable>
3. #include <queue>
4. template <typename T>
5. class ThreadSafeQueue
6. {
7. private:
8. size\_t \_capacity = 0;
9. std::queue<T> \_queue;
10. std::mutex \_mutex; // The mutex for locking the queue.
11. std::condition\_variable \_cond\_pop; // condition variables for pushing/popping.
12. std::condition\_variable \_cond\_push;
13. std::atomic<bool> \_shutdown; // Whether the queue is shutdown
14. };

* [Line 5-10] The ThreadSafeQueue Class is defined as a template class since it is uses a queue object of any arbitrary data type.
* Programs that use this queue can add and remove elements with single method calls add() and remove() without knowing the implementation details.

**Constructor**

The constructor simply initializes the queue capacity

ThreadSafeQueue(int capacity) : \_capacity(capacity), \_shutdown(false) {}

**Destructor**

The queue object is destroyed automatically.

~ThreadSafeQueue() {}

**Remove an Element**

The core of remove() method looks as follows:

void remove(T & item) {

std::unique\_lock<std::mutex> lock(\_mutex);

\_cond\_pop.wait(lock , [this] { return \_shutdown || !\_queue.empty(); });

* The method is protected by a mutex, which is locked using scope guard std::unique\_lock.
* If the queue is empty, we wait on the condition variable \_cond\_pop. This releases the lock to other threads and blocks until we are notified that the condition has been met. In this case, the call to wait()returns when queue has element to pop or on shutdown.

**Add an Element**

void add(const T& item) {

std::unique\_lock<std::mutex> lock(\_mutex);

\_cond\_push.wait(lock, [this] { return \_shutdown || \_queue.size() < \_capacity; }); // release lock and go join the waiting thread queue

// after the wait, we own the lock.

if (!\_shutdown)

{

\_queue.push(item);

lock.unlock();

// Wake up one popping thread.

\_cond\_pop.notify\_one();

}

* The condition notification will be provided when an element is pushed into the queue

to wake up threads that could be waiting on item.

* If the queue is full, we wait on the condition variable \_cond\_pop.

**Clear Method**

void clear() {

std::lock\_guard<std::mutex> lock(\_mutex);

while (!\_queue.empty()) {

\_queue.pop();

}

}

**Shutdown Method**

void shutdown() {

std::lock\_guard<std::mutex> lock(\_mutex);

\_shutdown = true;

//Notify all the consumers

\_cond\_pop.notify\_all();

//Notify all the producers

\_cond\_push.notify\_all();

}

**Class Diagram**

Condition

\_wait

\_signal

clear

shutdown

Shutdown Exception

Main Thread

1..\*

+ add()

+ remove()

+ clear()

+ shutdown()

shutdown

Producer Thread

1.. \*

1

empty

full

1

**ThreadSafeQueue**

* \_capacity
* \_mutex
* \_cond\_pop
* \_cond\_push
* \_shutdown

remove

add

Consumer Thread